VNIVERSIDAD NACIONAL
AVFNOMA DE
MEXICO

**UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO**

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

# "ARQUITECTURA DE LA HERRAMIENTA INTEGRAL PARA MoProSoft"

## T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN INGENIERÍA
(COMPUTACIÓN)

## P R E S E N T A :

HAFIZ ZURITA RENDÓN

DIRECTORA DE TESIS:
M. EN C. MARÍA GUADALUPE ELENA
IBARGÜENGOITIA GONZÁLEZ

México, D. F. 2005

# Agradecimientos

A mi familia, a mis amigos, a Lupita y a Hanna por el apoyo que me han brindado, por sus consejos y por convencerme.

A Borland que nos prestó el software necesario para realizar el proyecto HIM.

# Índice

# *Introducción*

El proyecto del cual forma parte la tesis tiene como finalidad construir una herramienta integral que apoye a las empresas mexicanas en la adopción de MoProSoft. Actualmente no existe en el mercado ninguna herramienta con las características que se proponen, como la distribución gratuita, y las que ofrecen funcionalidad parecida son costosas:

- Rational Clear Case de IBM.
- RCS de Component Software Inc.
- Adonis de Business Objectives Consulting.
- Primavera Project Planner de Primavera Systems, Inc.
- Merant Professional de Merant.
- BugBase de Threerock Software.
- ARP de RmyA.
- System Architect de Popkin Software.
- Mks Integrity Suite de MKS.
- Rational Unified Process de IBM.

## *Objetivo*

El objetivo de la tesis es especificar la arquitectura de la Herramienta Integral de MoProSoft (HIM). Para lograr esto se hace uso de patrones de arquitectura y diseño, apoyados en el lenguaje de modelado UML y en el Proceso Unificado. La tesis forma parte de un proyecto en el que participan otras cinco personas. Debido a esta división podría haber inconsistencias al momento de componer los módulos que conforman a HIM. Por esta razón la tesis tendrá un enfoque integrador describiendo todos los módulos y sus interrelaciones desde un punto de vista arquitectónico.

## *Alcance del Proyecto*

Esta tesis forma parte del proyecto dirigido por Guadalupe Ibargüengoitia y Hanna Oktaba para la especificación de la Herramienta Integral de MoProSoft (HIM). Debido a la magnitud de la empresa se decidió dividirla en 6 tesis, cada una de las cuales corresponde a un módulo de la arquitectura de HIM. A continuación se listan las tesis junto con sus responsables:

**Tutora:** Hanna Oktaba

| # | Nombre | Responsable |
|---|---|---|
| 1 | Utilización de patrones y la arquitectura J2EE para el diseño de la interfaz de usuario de la Herramienta Integral MoProSoft (HIM). | Karín Valdivieso |

| | | |
|---|---|---|
| 2 | Aplicación de patrones basados en J2EE para el diseño e implementación de la capa de control de la Herramienta Integral para MoProSoft (HIM). | Marcos Oscar Vázquez Morales |
| 3 | Análisis e Implementación de Esquemas de Seguridad Aplicados a la Herramienta Integral MoProSoft (HIM). | Jorge Cruz Vázquez |

**Tutora:** Guadalupe Ibargüengoitia

| # | Nombre | Responsable |
|---|---|---|
| 4 | Arquitectura de la Herramienta Integral para MoProSoft. | Hafiz Zurita Rendón |
| 5 | La Sincronización de los Elementos de una Base de Conocimiento para MoProSoft y su Aplicación en una Herramienta Integral. | Araceli Eugenia Mercado Fernández |
| 6 | Uso de la tecnología RDF para representar y manejar los procesos MoProSoft y su aplicación en HIM. | Ernesto Hernández Uribe |

## *Estructura*

El documento fue dividido en 5 capítulos:

- Requerimientos de HIM.
- Patrones de Diseño y Arquitectura.
- Patrones de la Arquitectura de HIM.
- Tecnologías y Estrategias para Patrones.
- Arquitectura de HIM.

En el capítulo sobre los Requerimientos de HIM se establecen las bases de la tesis. Incluye la descripción de MoProSoft, tratando las gestiones y procesos que el modelo maneja. También se describe el esquema de estos últimos, el cual además de ser utilizado para la documentación del modelo, es propuesto para documentar cualquier proceso existente o de nueva adopción dentro de la organización[1].

Posteriormente se plantea la importancia de HIM en la industria mexicana de software, fundamentada en parte por el elevado costo que implica la implantación de un estándar de calidad. También se describe la funcionalidad incluida en la herramienta, que consiste básicamente en la administración de la base de conocimiento (BC), en la creación del marco de trabajo para el usuario (procesos, proyectos, actividades y roles asignados), y en el almacenamiento de los productos generados. Debido a la importancia de la BC, es tratada a mayor detalle, describiendo la tecnología RDF en la cual se implementa el modelo de procesos así como algunos productos y la información sobre los usuarios.

[1] Empresa o área interna dedicada al desarrollo y/o mantenimiento de software [MoProSoft].

En la sección Análisis se incluyen los diagramas de clases que representan a las entidades identificadas en el modelo, junto con sus atributos y responsabilidades. Siguiendo el proceso de desarrollo, estas clases servirán como entrada para el diseño por lo cual es importante que sean especificadas claramente.

En el capítulo 2 se explican qué son los patrones, al principio de manera general para posteriormente centrarse en los correspondientes a la arquitectura de software. A fin de proporcionar una estructura estándar para su definición se presenta una plantilla, la cual incluye las tres partes de todo patrón: Contexto, Problema y Solución.

Debido a la gran variedad de patrones existentes en la actualidad y que tienen como finalidad atacar distintas etapas o niveles en el desarrollo de software, se incluye la descripción de las *categorías de los patrones*, como son patrones de arquitectura, patrones de diseño y modismos.

En el capítulo 3, una vez establecidas las bases de lo que es un patrón, se describe el que fue elegido para la arquitectura de HIM: el Model – *View* – *Controller* (MVC). Adicionalmente en cada uno de los componentes del MVC se usan otros patrones que apoyan la realización de tareas más específicas. Un ejemplo de esto son los patrones *View Helper* y el *Composite View* que permiten la construcción dinámica de las pantallas de usuario, en la capa de vista. De esta forma se plantean todos los patrones usados por la herramienta.

Como se expone en la descripción de los patrones, éstos sólo proponen una solución a un problema especifico en un contexto dado, pero no proporcionan la implementación de esa solución, para eso es necesario elegir un lenguaje de programación y una tecnología, sobre los cuales se plasme el patrón. Sobre estos temas trata el capítulo 4.

Debido a restricciones de la herramienta, como la portabilidad y el mantenimiento por sus propios usuarios[2], se eligió como lenguaje de programación Java, en conjunto con bibliotecas de terceros[3]. En algunos casos se hará uso de subsistemas que proveen gran parte de la funcionalidad requerida como es el caso del *Struts* utilizado en el control y en la vista. Para aprovechar mejor *Struts* se especifica de manera detallada cada uno de sus componentes, cómo se interrelacionan y las responsabilidades que cubren.

En el capítulo 5, se definen los elementos que constituirán la arquitectura de HIM. Este capítulo se estructura de acuerdo a las capas del patrón. En cada uno de los casos se presentan los diagramas a nivel diseño, definiendo las clases

[2] Especificadas en el documento "Requerimientos no Funcionales" del primer ciclo. Todos los documentos del primer y segundo ciclo del desarrollo de HIM son incluidos en el CD de la documentación del proyecto. Ver bibliografía.
[3] Por ejemplo, Jena de HP para el manejo de RDF.

principales y sus relaciones con clases de apoyo. También se especifican los patrones usados.

Finalmente se presentan las conclusiones obtenidas como resultado de la aplicación de patrones y su implementación con tecnologías de Java, RDF y *Struts*.

# *Capítulo 1*

# *Requerimientos de HIM*

Durante el curso de Ingeniería de Software Orientado a Objetos de la Maestría en Ciencias e Ingeniería de Computación se debía realizar un proyecto en el cual se aplicara la tecnología orientada a objetos buscando lograr las normas de calidad planteadas por la ingeniería de software. La primer cuestión que surgió fue qué modelo de calidad se usaría para llevar a cabo el desarrollo. Además se tenía que constituir una empresa en el sentido de establecer los procesos que indican los modelos y formar equipos de trabajo donde sus integrantes cumplieran con roles específicos.

No resultó difícil la elección del modelo de calidad ya que en esa época se había liberado la versión 1.1 de MoProSoft. Este modelo fue desarrollado por solicitud de la Secretaría de Economía para servir de base a la Norma Mexicana para la Industria de Desarrollo y Mantenimiento de Software. Con MoProSoft además de cumplir con el objetivo del curso se tenía la situación inmejorable de usar un modelo concebido especialmente para la industria mexicana.

Hecho lo anterior sólo restaba elegir el proyecto a desarrollar. Al analizar los modelos de calidad se reconoció que algo que dificulta la adopción de éstos es el gran número de actividades que se deben llevar a cabo sin tener una guía, aparte del documento del modelo. Esta guía debería indicar las actividades a realizar, el orden para llevarlas a cabo, los roles responsables, aparte de hacerlo de manera automatizada. Para lograrlo se planteó una herramienta informática que podría apoyar la ejecución de las actividades de manera que el usuario se enfocará en llevarlas a cabo sin preocuparse por las restricciones de dependencias, roles y documentos, ya que la herramienta lo indicaría. Fue entonces cuando surgió la Herramienta Integral para MoProSoft: HIM.

Habiendo definido el modelo de procesos y el proyecto sólo restaba ponerse a trabajar, fue entonces cuando se tuvo la primer dificultad. Se tenían que adoptar los procesos de MoProSoft para hacer un sistema que apoyara la aplicación de esos mismos procesos. Fue como escribir el compilador de un lenguaje de programación con el mismo lenguaje de programación. Para resolver esto se creó la especificación de la herramienta y se eligieron los procesos que se usarían en el desarrollo. Con estos primeros pasos se aclaró el panorama, pero todavía hacia falta determinar quiénes cumplirían los roles, establecer un plan de trabajo y definir mecanismos para el seguimientos de la actividades y el cumplimiento de los objetivos.

Estas incógnitas fueron resueltas haciendo uso del Proceso Unificado, del *Personal Software Process* y el *Team Software Process*, y por supuesto de MoProSoft.

Del Proceso Unificado se tomaron los flujos de trabajo, que fueron usados para complementar el proceso de "Desarrollo y Mantenimiento de Software" de MoProSoft. Del *Personal Software Process* y del *Team Software Process* se

utilizaron las formas de registro de tiempo, los reportes de actividades y el manejo de riesgos. De MoProSoft se utilizaron todos sus procesos, a partir de los cuales se crearon el grupo directivo, los grupos de administración de proyectos y los grupos de desarrollo.

Es importante mencionar que el desarrollo de la herramienta se dividió en dos ciclos, en el primero se debía crear la especificación de HIM hasta un nivel de diseño y en el segundo se revisaría este último, depurándolo y verificando que los artefactos creados, desde casos de uso hasta clases del diseño, contemplaran la funcionalidad requerida. Una vez definido lo anterior se implementaría un primer prototipo con funcionalidad básica.

## *MoProSoft*

Citado textualmente del prólogo de MoProSoft:

"El presente documento fue desarrollado a solicitud de la Secretaria de Economía para servir de base a la Norma Mexicana para la Industria de Desarrollo y Mantenimiento de Software bajo el convenio con la Facultad de Ciencias, Universidad Nacional Autónoma de México."

Teniendo tal petición, el equipo redactor se planteó el propósito de crear un modelo de procesos que fomente la estandarización en la operación de la industria de software. Esto incluye la incorporación de las mejores prácticas en gestión e ingeniería de software. Además se pretende que la adopción del modelo permita evaluar la capacidad de las organizaciones, para ofrecer servicios con calidad y alcanzar niveles internacionales de competitividad.

MoProSoft es un modelo que pretende ser fácil de entender y aplicar. Debido a que su adopción no es costosa, sirve de base para alcanzar evaluaciones exitosas con otros modelos como ISO 9000:2000 o CMM®. Los criterios empleados para su elaboración fueron:

- Generar una estructura de procesos que coincida con la estructura de las organizaciones de la industria de software: Alta Dirección, Gestión y Operación.
- Destacar el papel de la Alta Dirección en la planeación estratégica, como promotor del buen funcionamiento de la organización.
- Considerar a la Gestión como proveedor de recursos, procesos y proyectos, así como responsable de vigilar el cumplimiento de los objetivos estratégicos de la organización.
- Considerar a la Operación como ejecutor de los proyectos de desarrollo y mantenimiento de software.

- Destacar la importancia de la gestión de recursos, en particular los que componen la base de conocimiento de la organización.

- Basar el modelo de procesos en ISO9000:2000 y nivel 2 y 3 de CMM® v1.1. Usar como marco general ISO/IEC 15504 – Software Process Assesment e incorporar las mejores prácticas de otros modelos de referencia tales como PMBOK, SWEBOK, entre otros.

## *Estructura de MoProSoft*

MoProSoft como modelo que guía a empresas en el proceso completo de desarrollo de software, cuenta con una estructura definida en un patrón de procesos, de manera que su entendimiento y posterior aplicación se facilite. Además si la empresa ya cuenta con procesos, o necesita implantar alguno no previsto, podrá seguir el patrón, de manera que se garantizará una documentación completa, clara y consistente con la del estándar.

El patrón está constituido por tres partes[4]:

- **Definición general del proceso.**
  - Identifica su nombre.
  - Categoría a la que pertenece.
  - Propósito.
  - Descripción general de sus actividades.
  - Objetivos, indicadores y metas cuantitativas.
  - Responsabilidad y autoridad.
  - Subprocesos, en caso de tenerlos.
  - Procesos relacionados.
  - Entradas, salidas y productos internos.
  - Referencias bibliográficas.

- **Prácticas.**
  - Roles involucrados y la capacitación requerida.
  - Descripción de las actividades en detalle, asociándolas a los objetivos.
  - Diagrama de flujo de trabajo.
  - Verificaciones y validaciones.
  - Productos que se incorporan a la base de conocimiento.
  - Infraestructura necesaria para las actividades.
  - Mediciones.
  - Prácticas para la capacitación.
  - Manejo de situaciones excepcionales.
  - Uso de lecciones aprendidas.

[4] Para mayor información consultar la sección *Patrón de Procesos* en el documento de MoProSoft versión 1.1, mayo 2003.

- **Guías de ajuste.**
  - Modificaciones al proceso que no deben afectar los objetivos del mismo.

## *Estructura del modelo de procesos*

El modelo de procesos cuenta con tres categorías de procesos que reflejan la estructura de una organización:

- **Alta Dirección.**
  Aborda las prácticas relacionadas con la gestión del negocio. Proporciona los lineamientos a los procesos de la Categoría de Gestión y se retroalimenta con la información generada por ellos.
- **Gestión.**
  Aborda las prácticas de gestión de procesos, proyectos y recursos. Proporciona los lineamientos para el funcionamiento de los procesos de la Categoría de Operación, recibe y evalúa la información generada por éstos y comunica los resultados a la Categoría de Alta Dirección.
- **Operación.**
  Aborda las prácticas de los procesos de desarrollo y mantenimiento de software. Realiza sus actividades de acuerdo a los elementos proporcionados por la Categoría de Gestión y entrega a ésta la información y productos generados.

En el siguiente diagrama (ver figura 1.1) se presentan los procesos y subprocesos de cada una de las categorías.

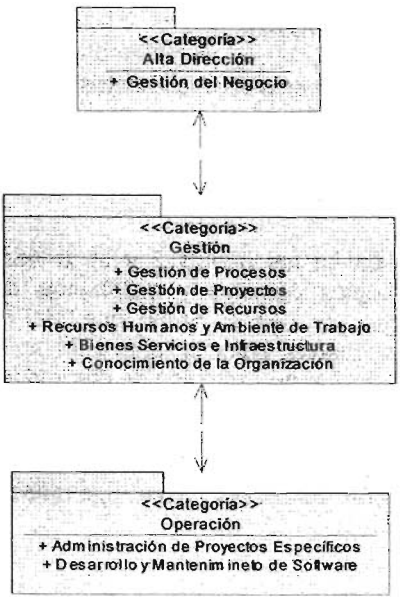



**Figura 1.1**

## *HIM*

Actualmente, en México, muy pocas de las empresas que desarrollan software aplican procesos. Esta situación se debe, en muchos de los casos, al elevado costo que implica su implantación, en términos de tiempo y recursos. Es por ello que resulta de suma importancia facilitar la adopción de procesos, proveyendo a las empresas con una herramienta que las guíe a través de los incluidos en MoProSoft.

En respuesta a esta necesidad surgió la Herramienta Integrada para MoProSoft (HIM). HIM tiene como objetivo representar fidedignamente el contenido de MoProSoft añadiendo funcionalidad propia de un sistema de software, como la aplicación de restricciones en la realización de las actividades, el almacenaje de documentos, la administración de usuarios y la notificación de cambios. Esta herramienta está basada en los procesos propuestos por el modelo, y no en los productos generados.

En el documento "Descripción del Proyecto" del primer ciclo del desarrollo de HIM se especifican los requerimientos generales de la herramienta:

1. Que forme un ambiente integrado para soportar las actividades de MoProSoft.
2. Que use la tecnología que permita su distribución como software libre.

3. Que tenga las siguientes cualidades:
   a. Portabilidad.
   b. Soporte de diversos formatos de documentos.
   c. Seguridad en el resguardo y en el acceso.
   d. Usabilidad.
   e. Soporte de grupos de trabajo.
   f. Mantenible por sus propios usuarios.

Como se especifica en los puntos 2 y 3f la herramienta deberá poder mantenerse por sus propios usuarios y distribuirse de manera libre. Estas características tienen que ver con el hecho que MoProSoft está enfocado a empresas medianas y pequeñas y por lo tanto HIM no debe implicar un gasto oneroso, lo cual podría dificultar que este tipo de empresas la usaran.

Estas características además de servir para establecer los requerimientos no funcionales fueron determinantes para la elección de las tecnologías que se proponen en el diseño, como son *Struts* y Jena, que más adelante serán tratadas.

## *Funcionalidad de HIM*

El alcance de esta tesis se fijó hasta un nivel de diseño, para lo cual se hace uso de patrones documentados a detalle. Adicionalmente se construyó un primer prototipo con funcionalidad básica que permitió la validación del diseño. Cierta funcionalidad se reconoció como necesaria pero no fue incluida en ninguna tesis, sin embargo se tuvo cuidado que estos puntos, propuestos como temas para trabajos futuros puedan integrarse con transparencia sin tener que rediseñar lo ya existente.

La herramienta, en su primera versión incluirá lo siguiente:

- Acceso al contenido del modelo de procesos.
- Presentar al usuario un marco de trabajo con los elementos de las actividades (descripción, objetivos, productos, etc), dependiendo de los roles que tiene asignados.
- Establecer las dependencias incluidas en los procesos, de manera que el usuario no pueda eludir las restricciones que representan.
- Almacenar los productos generados.
- Comunicar al usuario el estado de las actividades y sus productos.
- Permitir la explotación de la información generada.

La funcionalidad que se dejó pendiente es:

- Notificar el cambio de estado de los productos.

- Manejo de repositorios personales.
- Administración de respaldos.
- Soporte para el acceso concurrente.
- Búsquedas para explotar la información de los Productos de Información Especializada.
- Manejo de documentos del tipo "caja negra".

## *Base de Conocimiento*

La Base de Conocimiento constituye la médula de HIM, ya que en ella se guardan el modelo de procesos, los productos y se construyen los objetos del negocio, proveyendo todos los datos necesarios al resto de la aplicación.

Para implementar la persistencia se optó por el *Resource* Description *Framework* (RDF) [Hernández]. Esta tecnología forma parte de una iniciativa para estructurar la información de Internet de manera que las búsquedas sean realizadas por sistemas de manera automatizada. Para lograr esto, RDF incluye la capacidad de realizar inferencias a partir de los datos que son registrados, lo cual habilita a un sistema para extender dinámicamente su información a través de nuevas relaciones. Los mecanismos en los cuales se basa usan reglas de la lógica de primer orden. Esta tecnología está implantada en XML lo cual asegura su portabilidad e independencia de cualquier infraestructura usada. Para poder conectar RDF con el resto de la aplicación se eligió Jena, una API en Java, desarrollada por Hewlet Packard.

Utilizar esta tecnología incipiente hace que la herramienta sea innovadora. RDF es usado para instituir el modelo de procesos de MoProSoft de forma que se recuperen y apliquen las reglas contenidas en el documento a las tareas que facilita HIM. Además, RDF se usa para almacenar el estado de la aplicación lo que incluye datos de los documentos generados, de los usuarios y de las actividades, entre otros.

La persistencia está además formada por un repositorio apoyado en el sistema de archivos de la máquina servidor. Su función principal es almacenar ciertos documentos, conocidos como Cajas Negras. En el repositorio únicamente se guarda el contenido del documento, mientras que información como su nombre, versión, formato, etc. son registrados en RDF. Debido a esta dicotomía en la persistencia se integrará un módulo de sincronización que permita la actualización de los componentes que la integran, de esta manera cualquier cambio es reflejado automáticamente a las partes afectadas [Mercado].

En el caso de empresas que ya cuentan con una base de conocimiento, en una base de datos relacional o en cualquier otro sistema; podrán integrarla

fácilmente a HIM implementando una interfaz que defina las responsabilidades esperadas por la aplicación.

## *Análisis*

### *Casos de Uso*

Como se menciona en el Proceso Unificado, el modelo de casos de uso es un artefacto que permite a los desarrolladores y a los clientes llegar a un acuerdo sobre la funcionalidad que el sistema deberá proveer, es decir, posibilita la especificación de los requerimientos. Una vez que el modelo de casos de uso ha sido madurado se inicia el flujo del análisis, del cual constituye la entrada fundamental. Posteriormente también será usado en el diseño y las pruebas.

En la revisión de los procesos de MoProSoft se observó que algunas de las actividades no serían apoyadas por la herramienta, debido principalmente a limitaciones de alcance en esta versión. No obstante, para tener un mejor entendimiento del negocio se realizó un modelo de casos de uso de MoProSoft, a partir del cual se determinó qué funcionalidad incluirá HIM, y que será representada en el modelo de casos de uso de la herramienta a nivel de requerimientos. Este modelo es el mostrado en la presente sección y con él se busca sentar las bases del análisis y del diseño

Una vez obtenidos estos modelos, se decidió validar si realmente los requerimientos estaban representados en los casos de uso, en las clases del análisis y del diseño. Para lograrlo se generó el documento Registro de Rastreo[5], que de manera resumida muestra como están relacionados estos artefactos.

#### Diagramas de casos de uso

#### Diagrama General

A continuación se muestran los casos de uso sustantivos de HIM, figura 1.2. Algunos de ellos incluyen subcasos que brindan un mayor detalle de la funcionalidad representada. En cada diagrama se especifican los roles de los actores.

[5] Ver el apéndice.

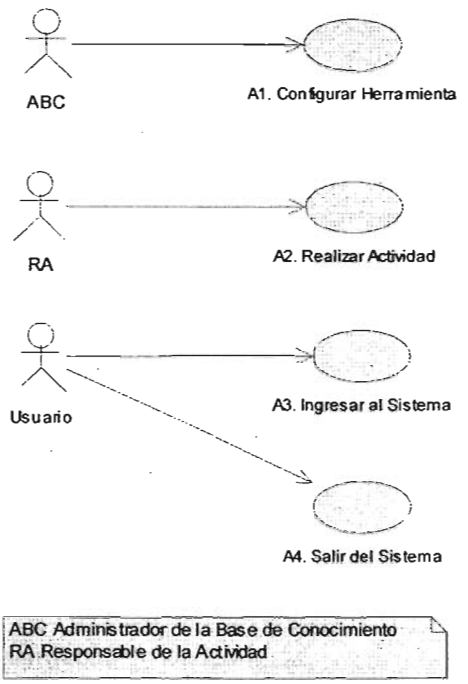



**Figura 1.2**

### Caso de Uso: A1. Configurar Herramienta

Incluye lo necesario para llevar acabo la configuración de la herramienta con respecto a la Base de Conocimiento (BC), ver figura 1.3. La funcionalidad es la siguiente:

| Identificador | Descripción |
|---|---|
| A1.1.1 | Permite al Administrador de la BC (ABC) especificar donde será instalada la BC. Esto es de suma importancia ya que las referencias entre los elementos del marco de trabajo (almacenados en RDF) dependen de esta ruta. |
| A1.1.2 | Genera los directorios necesarios para almacenar los archivos RDF, los repositorios personales, las bibliotecas, etc. |
| A1.1.3 | La creación del marco de trabajo incluye la instalación de todo lo referente a la representación en RDF de MoProSoft. |
| A1.1.4 | Permite la creación de una cuenta para el ABC. |

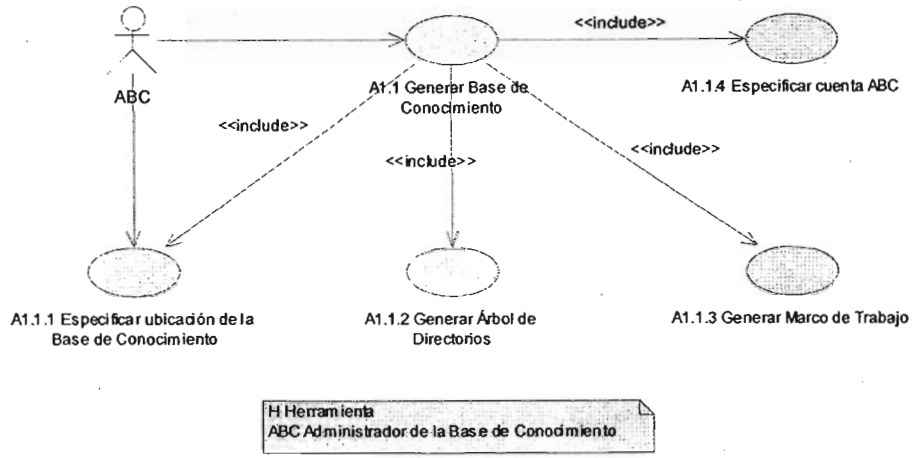



**Figura 1.3**

## Caso de Uso: A2. Realizar Actividad

La realización de actividades se reduce a la creación y/o modificación de productos (documentos), ver figura 1.4.

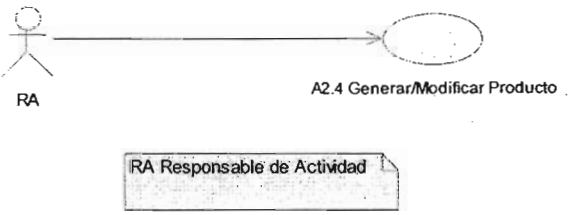



**Figura 1.4**

## Caso de Uso: A2.4 Generar / Modificar Producto

El sistema proporciona la funcionalidad necesaria para manejar los productos (ver figura 1.5), los cuales pueden ser de dos tipos: Cajas Negras y Productos de Información Especializada (PIE). Los primeros son documentos producidos en editores distintos de HIM. La herramienta sólo se encarga de almacenarlos, de relacionarlos con la actividad en la cual están definidos y de administrar datos descriptivos, como el nombre, la versión y la ruta en el repositorio. Los PIEs son manejados completamente por HIM abarcando desde su captura hasta explotar los datos que contienen. La funcionalidad es la siguiente:

| Identificador | Descripción |
|---|---|
| A2.4.2 | Incluye la obtención de un producto junto con sus dependencias, y la capacidad de mostrarlo. Esta funcionalidad de edición sólo aplica a los productos caja negra y consiste en desplegarlos en la aplicación con la que fueron generados. Por ejemplo, si el Plan de Manejo de Riesgos de un proyecto fue generado en Microsoft Word, y posteriormente se solicita el documento para editarlo, HIM invocará a Word y éste se encargará de presentarlo. |
| A2.4.3 | Tiene que ver con el almacenamiento de los productos en el repositorio. Esto además conlleva la actualización del estado del producto y la notificación a los roles interesados de este cambio. |
| A2.4.4 | Permite la verificación / validación de los productos que así lo requieran, por los roles especificados en MoProSoft. |
| A2.4.5 | Incluye la explotación de los PIEs: captura, modificación, consulta, búsqueda de información e impresión. |

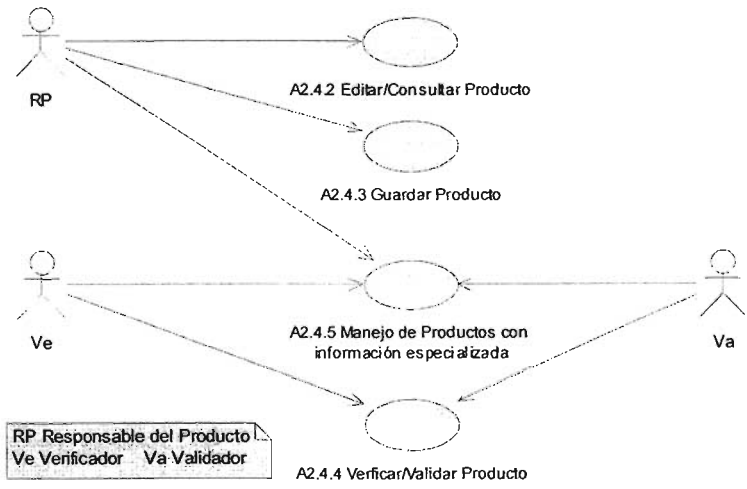



**Figura 1.5**

**Nota**: El caso de uso A2.4.1 "Consultar Producto" se fusionó con el A2.4.2 "Editar Producto", debido al alcance del sistema.

### Caso de Uso: A2.4.2 Editar / Consultar Producto

En la mayoría de las actividades de MoProSoft se requiere generar un producto, para lo cual es necesario cubrir las dependencias que pueda tener.

Además la herramienta permitirá desplegar los productos para poder ser consultados y editados (ver figura 1.6). La funcionalidad es la siguiente:

| Identificador | Descripción |
|---|---|
| A2.4.2.1 | Permite la búsqueda de los productos, obteniendo la información que los describe, el estado y el contenido; el cual se localizará en el repositorio o en RDF dependiendo del tipo de producto. |
| A2.4.2.2 | Obtiene las dependencias de los productos, tanto de entrada (productos que son necesarios para la realización del actual) como de salida (productos que dependen del actual para ser generados). Estas dependencias deberán ser mostradas para que el usuario pueda conocer los producto necesarios / afectados en la realización de alguno de su interés. |
| A2.4.2.3 | Este caso de uso sólo aplica a los productos Cajas Negras y consiste en registrar el programa que despliega y permite modificar el producto. De esta forma al solicitarlo HIM deberá invocar al programa adecuado que lo muestre. |

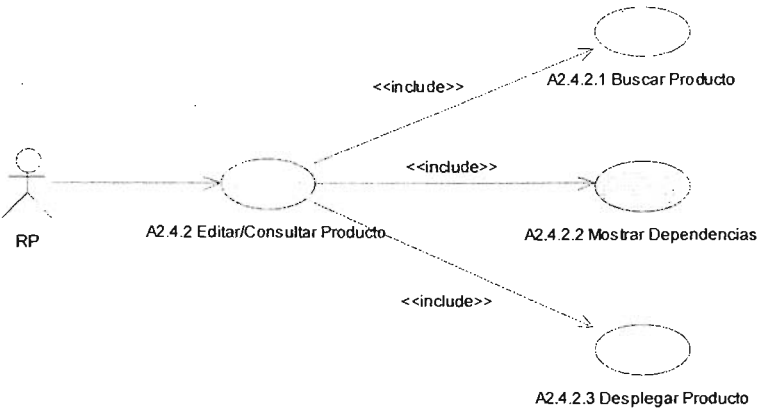



**Figura 1.6**

### Caso de Uso: A2.4.3 Guardar Producto

Este caso de uso maneja el almacenamiento de los productos, lo cual implica un cambio de estado (terminado, verificado, validado, etc.) dependiendo la etapa por la que este pasando. Los cambios de estado deberán ser notificados a los usuarios responsables de manera que lleven a cabo la actividad que les fue asignada (ver figura 1.7). La funcionalidad es la siguiente:

| Identificador | Descripción |
|---|---|
| A2.4.3.3 | La herramienta almacenará los productos en un repositorio común de manera que puedan ser recuperados por cualquier usuario que los requiera. Adicionalmente se tiene planeado proporcionar repositorios individuales en los cuales los usuarios puedan guardar los productos mientras están trabando en ellos. |
| A2.4.3.4 | Cada vez que un producto es modificado cambia su estado lo cual deberá ser registrado. En base a los estados, la herramienta tomará decisiones sobre las actividades que son aplicables al producto. |
| A2.4.3.6 | En ciertos casos deberá notificarse a los usuarios los cambios de estado que sufren los productos que tienen asignados. |

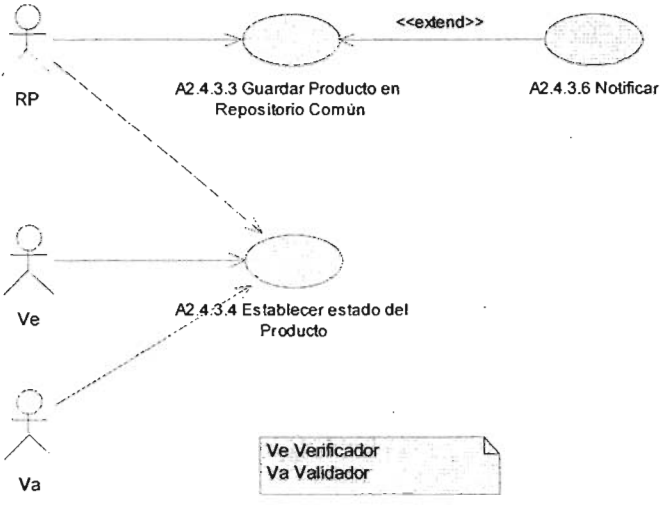



**Figura 1.7**

**Nota**: Los siguientes casos de uso se eliminaron por estar fuera de alcance:

- A2.4.3.1 "Guardar Producto Repositorio Privado".
- A2.4.3.2 "Guardar Producto Repositorio Grupo".
- A2.4.3.5 "Respaldar Producto".

### Caso de Uso: A2.4.5 Manejo de Productos con Información Especializada

Los Productos con Información Especializada (PIEs) son productos que fueron elegidos para ser manejados por HIM de manera que su contenido pudiera ser explotado (ver figura 1.8). El manejo de la información de los PIEs posibilita la creación de la base de conocimiento. Un ejemplo lo constituyen las lecciones aprendidas, cuya información puede ser consultada cuando se enfrenta un

problema, que resulta similar a alguno que ya fue resuelto. La funcionalidad es la siguiente:

| Identificador | Descripción |
|---|---|
| A2.4.5.1 | La herramienta permitirá la captura de los PIEs. |
| A2.4.5.2 | La herramienta permitirá modificar los PIEs. |
| A2.4.5.3 | La consulta sólo incluye desplegar el contenido de un PIE, en particular para su revisión. |
| A2.4.5.4 | La herramienta permitirá la impresión del PIE en un formato definido en una plantilla. |
| A2.4.5.5 | La búsqueda se realiza basándose en criterios especificados por el usuario, y es llevada a cabo sobre el contenido de los PIEs que cumplen con los criterios. |

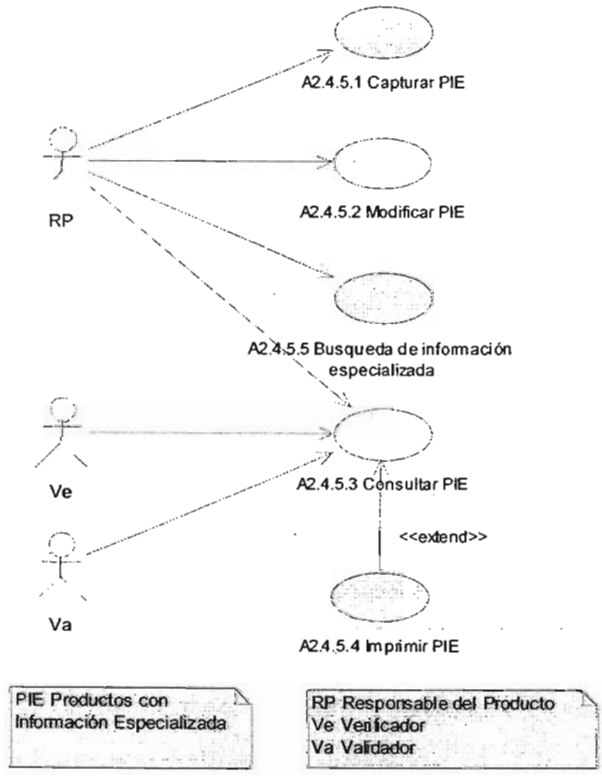



**Figura 1.8**

### Caso de Uso: A3 Ingresar al Sistema

El ingreso al sistema implica bastante más que la autenticación del usuario. Una vez que el usuario ingresa se debe obtener su marco general que incluye los roles que tiene asociados, y los procesos y proyectos en los cuales aparecen esos roles (ver figura 1.9). Esta información es desplegada y el usuario puede navegar entre ella hasta la actividad que requiera efectuar. La funcionalidad es la siguiente:

| Identificador | Descripción |
|---|---|
| A3.1 | Incluye la recuperación del nombre de usuario y de la contraseña la cual se guarda encriptada. Se aplican los métodos de validación y dependiendo de su resultado se le presenta al usuario su marco de trabajo, o se le indica el error en la autenticación. |
| A3.2 | Los roles del usuario son obtenidos de archivos RDF, y sirven para conseguir los procesos y proyectos en los cuales participa. |
| A3.3 | Obtiene todo los proyectos en los cuales el usuario podrá realizar alguna actividad. |
| A3.4 | Obtiene todo los procesos en los cuales el usuario podrá realizar alguna actividad. |

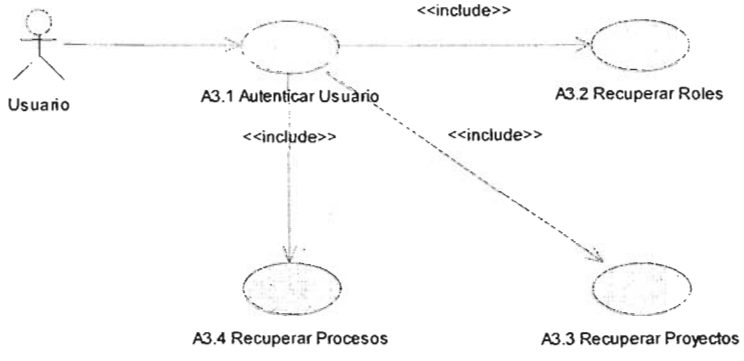



**Figura 1.9**

### Caso de Uso: A4 Salir del Sistema

Es necesario que cuando el usuario interrumpe su actividad indique al sistema que saldrá (ver figura 1.10). De esta forma se liberan los recursos asociados a su sesión y sus datos quedan en un estado consistente. La funcionalidad es la siguiente:

| Identificador | Descripción |
|---|---|
| A4.1 | Indica al sistema que el usuario no continuará trabajando, y que sean liberados los recursos que tenía asociados. |


**Figura 1.10**

## *Clases del análisis*

Estas clases representan una abstracción de las entidades identificadas tanto en este flujo como en la captura de requisitos, además servirán como base para clases o subsistemas que serán tratados en flujos de trabajo posteriores. El análisis se enfoca en requisitos funcionales posponiendo los no funcionales, que en este nivel se denominan como requisitos especiales. Este enfoque hace que las clases de análisis sean más evidentes en el dominio del problema y representen una granularidad mayor que sus contrapartes del diseño e implementación.

Las clases a este nivel (ver figura 1.11) son obtenidas a partir de los casos de uso[6]. Los elementos de interés son las responsabilidades, los atributos y las relaciones, a partir de los cuales se identifican clases de entidad y control. A continuación listamos cada una de ellas:

| Clase | Descripción |
|---|---|
| 1. MarcoGeneral | Se usa para encapsular los procesos y los proyectos a los que está vinculado un usuario. |
| 2. Proyecto | Contiene las actividades de un proyecto y los roles asociados. |
| 3. Proceso | Contiene las actividades de un proceso y los roles asociados. |
| 4. Rol | Encapsula los datos de un rol. |
| 5. Actividad | Contiene los datos sobre una actividad como son una descripción y un identificador. Además se incluyen los productos que la actividad maneja. |
| 6. Usuario | Encapsula los datos del usuario incluida la contraseña. |
| 7. Producto | Incluye datos del producto tanto para su manejo interno como los mostrados al usuario. También contiene referencias a los productos de los cuales depende, y hacia aquellos que dependen de él. |

[6] Para mayor detalle ver el documento Detalle de los Casos Uso, incluido en el CD de HIM.

| | |
|---|---|
| 8. Notificacion | Encapsula datos necesarios para la notificación, como son los destinatarios, el mensaje y el producto afectado. |
| 9. BaseConocimiento | Define las responsabilidades de la base de conocimiento. Además es un punto de control para las demás clases. |

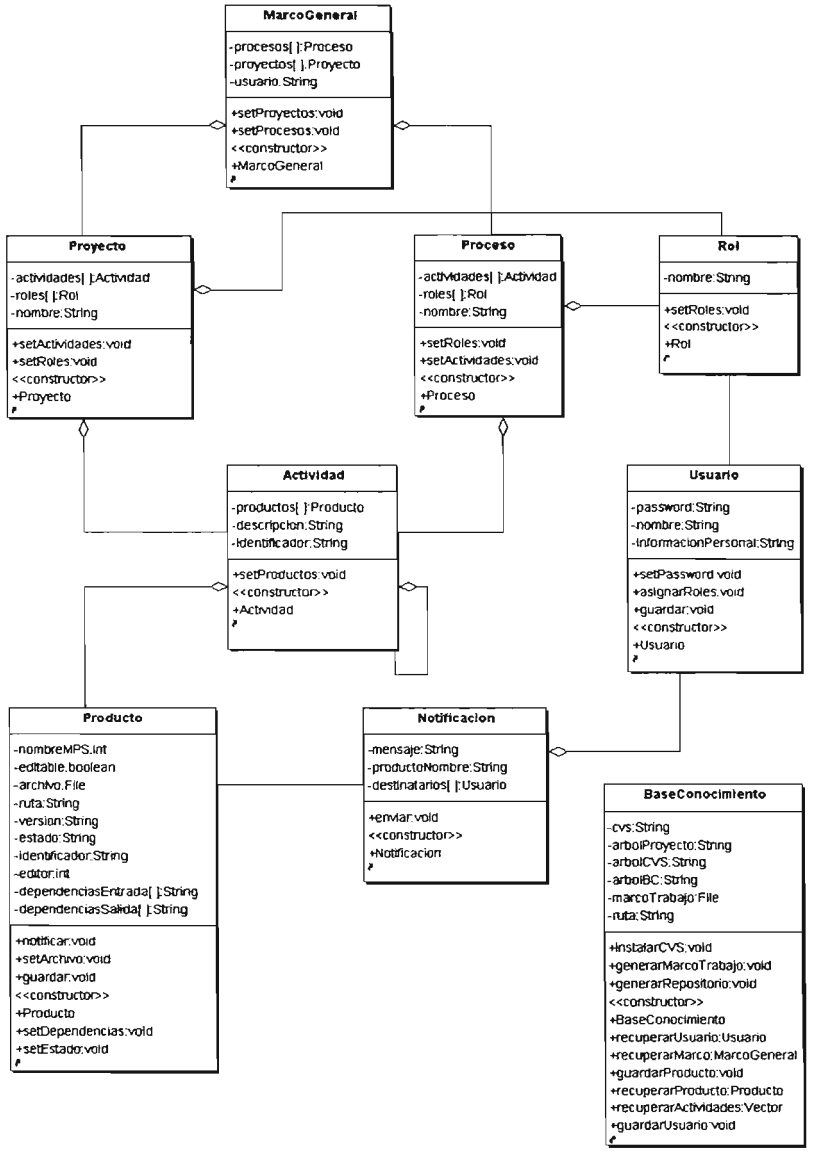



**Figura 1.11**

# *Capítulo 2*

# *Patrones de Diseño y Arquitectura*

"Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo siquiera dos veces de la misma forma".

**Christopher Alexander**

Este capítulo inicia respondiendo a la pregunta *¿qué es un patrón*? para lo cual se brinda una descripción de los mismos, analizando cómo surgen y su utilidad en el área de interés: la Arquitectura de Software. Esta primera sección se cierra con la definición proporcionada en el libro "*A System of Patterns*" de Buschmann [Buschmann], que engloba las características identificadas. Posteriormente se define la estructura de un patrón, estableciendo un esquema general que pretende dar un acercamiento al conocimiento representado, de manera que pueda ser de utilidad para quien lo consulta.

Después son tratadas las categorías de los patrones. Dado el creciente número de patrones que se manejan en la actualidad, y la diversidad de problemas que pretenden resolver, se ha hecho necesario catalogarlos. Gracias a esto es posible determinar de manera más rápida el patrón que pueda resolver un problema dado.

Para finalizar, se trata la descripción de los patrones, con el objetivo de refinar el esquema general propuesto, y establecer una plantilla detallada con los elementos que debe contener un patrón. Esta plantilla fue tomada sin cambios del libro [Buschmann], para que sirva como marco de referencia para la posterior definición de los patrones en el siguiente capítulo: Patrones para la Arquitectura de HIM.

## *¿Qué es un patrón?*

Cuando un grupo de expertos trabajan en un problema particular, no es común que lo *afronten* inventando una solución nueva. Es preferible recordar la solución dada a un problema similar, y reutilizar su esencia para resolver el nuevo problema. Este tipo de "comportamiento experto" se forma al pensar en el par problema – solución, común en diferentes dominios, tales como la arquitectura, la economía y la ingeniería de software.

Abstrayendo los elementos que intervienen en un par problema – solución específico, y refinando los factores comunes, se obtiene un patrón. En su libro "*The Timeless Way of Building*" [Alexander], el arquitecto Christopher Alexander define el término patrón de la siguiente manera:

> Cada patrón es una regla de tres partes, que expresa la relación entre un cierto contexto, un problema y su solución.
>
> Como un elemento en el mundo, cada patrón es una relación entre cierto contexto, un sistema de fuerzas, el cual ocurre repetidamente en ese contexto, y cierta configuración espacial, que permite que esas fuerzas sean resueltas por si mismas.
>
> Como un elemento del lenguaje, un patrón es una instrucción, la cual muestra cómo esa configuración espacial puede ser usada, una y otra vez, para resolver el sistema de fuerzas dado, en el momento en que el contexto es relevante.
>
> El patrón es, en resumen, al mismo tiempo la cosa que sucede en el mundo y la regla que indica cuándo y cómo crearla.

En la arquitectura de software también se pueden encontrar muchos patrones. Los expertos en ingeniería de software conocen estos patrones gracias a su experiencia práctica, y los siguen en el desarrollo de aplicaciones con características específicas.

Los siguientes puntos se identifican como propiedades de los patrones para la arquitectura de software:

- Identifican un problema de diseño recurrente, que surge en una situación específica, y presentan una solución a este.
- Documentan experiencias en el diseño, que han probado ser exitosas; permitiendo la reutilización del conocimiento de expertos.
- Identifican y especifican abstracciones en un nivel superior a clases simples, instancias o componentes. Típicamente un patrón describe varios componentes, clases u objetos, y detalla sus responsabilidades y relaciones.
- Proveen de un vocabulario común y entendimiento de los principios de desarrollo.
- Son medios de documentación de arquitecturas de software, ya que permiten describir la visión que el diseñador tiene al idear un sistema. También evita que otros alteren esta visión cuando se modifique o extienda la arquitectura original.
- Soportan la construcción de software con propiedades definidas, a través de un esquema de comportamiento funcional.

- Ayudan en la construcción de arquitecturas de software complejas y heterogéneas.

No obstante, que los patrones determinan la estructura básica de una solución a un problema de diseño particular, no definen una solución detallada. Un patrón provee un esquema para una solución genérica a una familia de problemas, en lugar de proporcionar módulos prefabricados que pueden ser usados directamente. Es responsabilidad del equipo de trabajo implementar el esquema de acuerdo a las necesidades concretas del problema de interés.

Como conclusión de esta primera parte en la definición de los patrones, se incluye la siguiente definición de Frank Buschmann [Buschmann]:

> Un patrón para la arquitectura de software, describe un problema de diseño particular y recurrente, que surge en un contexto específico, y presenta un esquema probado para su solución. El esquema solución es definido a través de la descripción de sus componentes, sus responsabilidades y relaciones, y la forma en la cual colaboran.

## *Estructura de un patrón*

A partir de lo visto en la sección anterior podemos establecer un esquema formado por tres partes subyacentes en todo patrón.

- *Contexto*: Situación donde surge el problema.
- *Problema*: El problema recurrente surgiendo en el contexto.
- *Solución*: Solución probada al problema.

El esquema como un todo denota una regla que establece la relación entre los tres elementos anteriores, los cuales mantienen un acoplamiento fuerte. No obstante definiremos qué se espera de cada una de estas tres partes.

### *Contexto*

Extiende la dicotomía problema – solución, describiendo la situación en la cual ocurre el problema. El contexto de un patrón puede ser bastante general, y a la vez servir para relacionar distintos patrones. Es prácticamente imposible determinar todas las situaciones, generales y específicas, en las cuales un patrón puede ser aplicado. Un enfoque puede ser listar los escenarios donde un problema manejado por un patrón puede ocurrir.

### *Problema*

Esta parte del esquema describe un problema que surge repetidamente en un contexto dado. Comienza con una descripción general, capturando la esencia del problema y después es completado por un conjunto de fuerzas.

Es importante definir qué entender por "fuerzas". Al mencionar las fuerzas y ligarlas con una configuración espacial C. Alexander [Alexander] brinda una visión ligada a la arquitectura, en la cual las fuerzas son constituidas por elementos físicos tales como el viento, el agua, los tipos de materiales, la altura, etc. Esto es válido para un inmueble pero ¿qué significado tienen las fuerzas en un sistema de software? El software, aun siendo algo intangible tiene restricciones impuestas por el medio en el que reside, por ejemplo la capacidad de almacenamiento y de memoria o las velocidades de transferencia de datos. Además existen restricciones debidas a otros sistemas de software, como la capacidad de una base de datos para procesar transacciones, o el número de procesos concurrentes que puede manejar un sistema operativo. Aunadas a las restricciones están las características propias del sistema, como el número de usuarios que puede atender, los tipos de archivos que deberá manejar o los mecanismos de seguridad que se deberán incluir.

Todos estos puntos constituyen las fuerzas que afectan a un sistema de software y que por lo tanto deberán manejarse de manera que la solución propuesta provea la funcionalidad requerida sin entrar en conflicto con ellas. En resumen el sistema de fuerzas está denotado por cualquier aspecto que debe ser considerado en la solución, tales como:

- Requerimientos, que la solución debe satisfacer.
- Restricciones.
- Propiedades deseables en la solución.

En general, las fuerzas tratan el problema desde varios puntos de vista y ayudan a entender sus detalles. Las fuerzas pueden complementarse o contradecirse. Dos fuerzas contradictorias son, por ejemplo, la extensibilidad de un sistema contra la reducción del tamaño del código. Pero lo más importante es que las fuerzas son la clave para resolver el problema, entre mejor balanceadas estén, se obtiene una mejor solución. Una discusión detallada de ellas es, por lo tanto, una parte esencial de la especificación del problema.

### *Solución*

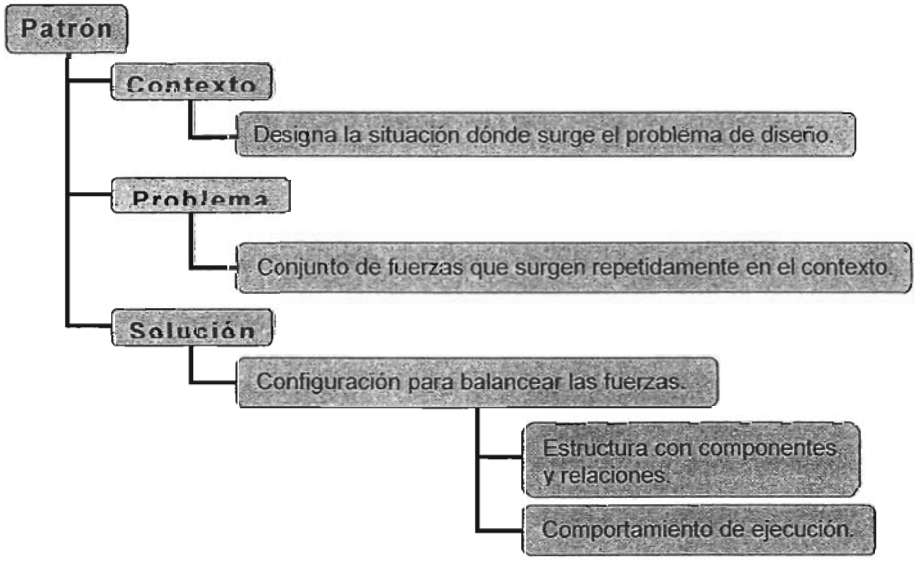
Esta parte muestra cómo resolver el problema recurrente, o mejor aún, cómo balancear las fuerzas asociadas con él. En la definición del *problema* se mencionó que las fuerzas están constituidas tanto por elementos del ambiente en el cual reside el sistema, como por las características o funcionalidad propia de

éste. Estos dos conjuntos de fuerzas pueden estar en conflicto desde que los recursos son limitados y los requerimientos de ellos se vuelven competitivos. Por esto la solución se da en parte al balancear las fuerzas. En la arquitectura del software la solución incluye dos aspectos:

- Todo patrón especifica cierta estructura, o configuración espacial de elementos. Esta estructura dirige los aspectos estáticos de la solución, y puede verse como una micro – arquitectura, formada de componentes y sus relaciones, donde los primeros sirven como bloques de construcción con una responsabilidad definida.
- Todo patrón especifica un comportamiento de ejecución, que dirige los aspectos dinámicos de la solución: ¿cómo se comunican y colaboran los componentes? ¿cómo se organiza el trabajo?

Es importante notar que la solución no resuelve necesariamente todas las fuerzas asociadas con el problema, en especial si éstas son contradictorias.

El siguiente diagrama resume el esquema tripartito:




## *Categorías de patrones*

Una revisión de los patrones revela que cubren varios rangos de escala y abstracción. Algunos ayudan a estructurar un sistema de software en subsistemas. Otros soportan el refinamiento de subsistemas y componentes, o las relaciones entre ellos. Otros más apoyan la implementación de aspectos de diseño

particulares, en lenguajes de programación específicos, e incluso hay patrones para la realización de pruebas.

Para refinar la clasificación, agruparemos los patrones en tres categorías:

- Patrones de arquitectura.
- Patrones de diseño.
- Modismos.

### *Patrones de arquitectura*

Una arquitectura de software viable es construida de acuerdo a principios estructurales globales, y estos principios son descritos en los patrones de arquitectura.

Como definición de un patrón de arquitectura incluimos la de Frank Buschmann [Buschmann]:

> Un patrón de arquitectura expresa el esquema de organización estructural fundamental para los sistemas de software. Provee de un conjunto de subsistemas predefinidos, especificando sus responsabilidades, e incluyendo reglas y guías para organizar las relaciones entre ellos.

Estos patrones constituyen plantillas para arquitecturas de software concretas. Especifican las propiedades estructurales del todo el sistema, y tiene impacto en la arquitectura de sus subsistemas. La selección de un patrón de arquitectura es, por lo tanto, una decisión fundamental y de alto nivel de abstracción en el desarrollo de un sistema de software.

### *Patrones de diseño*

Los subsistemas de una arquitectura de software, así como las relaciones entre ellos, comúnmente consisten de varias unidades arquitectónicas más pequeñas. Estas unidades son descritas usando los patrones de diseño. La definición del libro *Design Patterns* [Gamma] es la siguiente:

> Un patrón de diseño provee un esquema para refinar los subsistemas o los componentes de un sistema de software, o las relaciones entre ellos. Describe una estructura común recurrente de componentes intercomunicados que resuelve un problema de diseño general en un contexto particular.

Los patrones de diseño son patrones de mediana escala, y tienden a ser independientes de lenguajes o paradigmas de programación. La aplicación de un patrón de diseño no tiene efecto en la organización fundamental de un sistema de software, pero puede tener una fuerte influencia en la arquitectura de un subsistema. Muchos patrones de diseño proveen de una estructura para separar servicios o componentes en elementos menos complejos. Otros manejan la cooperación efectiva entre ellos.

### *Modismos*

Los modismos tratan con la implementación de asuntos particulares de diseño. De [Buschmann] se tiene la siguiente definición.

> Un modismo es un patrón de bajo nivel, específico para un lenguaje de programación. Describe cómo implementar aspectos particulares de un componente, o la relación entre ellos usando las características de un lenguaje dado.

Los modismos representan a los patrones de más bajo nivel, y manejan aspectos de diseño e implementación. La mayoría de los modismos son específicos de un lenguaje, ya que capturan la experiencia en programación existente. Frecuentemente el mismo modismo se ve diferente para distintos lenguajes de programación, y por lo tanto pueden ser útiles para un lenguaje y no tener sentido en otro. Por ejemplo, la comunidad de programadores de C++, usan modismos para la administración dinámica de recursos, en cambio el lenguaje Java está provisto de mecanismos para la recolección de basura, y por lo tanto no necesita de tales modismos.

## *Descripción de un patrón*

Los patrones deben ser presentados en una forma apropiada si queremos entenderlos y discutirlos. Una buena descripción ayuda a capturar la esencia del patrón inmediatamente: ¿cuál es el problema que el patrón maneja? ¿cuál es la solución que este propone? Una buena descripción también facilita los detalles necesarios para implementar el patrón, y considerar las consecuencias de su aplicación.

Los patrones deben ser descritos uniformemente, lo cual ayuda a comparar un patrón con otro, especialmente cuando se buscan soluciones alternativas a un problema. La estructura *contexto – problema – solución* que se trató anteriormente proporciona un formato que cumple con los requerimientos mencionados, ya que

captura las características esenciales de un patrón y facilita sus ideas claves [Buschmann]. Por lo tanto se usará esta estructura para definir una plantilla de descripción.

Inicialmente se debe considerar un nombre para el patrón, preferiblemente intuitivo, y que refleje la esencia del patrón. Esto facilitará compartirlo, discutirlo, y se convertirá en parte de nuestro vocabulario de diseño. Adicionalmente al nombre se pueden incluir alias mnemotécnicos que ayuden a simplificar las referencias al patrón.

Se añade un ejemplo introductorio que ayude a explicar el problema y las fuerzas asociadas. Este ejemplo será referido cuando se discuta la solución y los aspectos de implementación. Posteriormente se usan diagramas y escenarios para ilustrar los aspectos estáticos y dinámicos de la solución. Se inlcuyen guías que ayuden a transformar una arquitectura en la implementación del patrón. Junto con esto se agrega código, y se listan aplicaciones exitosas del patrón que realcen su credibilidad.

También se describen variantes del patrón, que proveen soluciones alternativas al problema. Estas variantes son descritas brevemente sin llegar al detalle del patrón original. Se incluye una discusión de los beneficios y desventajas de la aplicación del patrón, ayudando a decidir si el patrón ofrece una solución real. Apoyando esto, se ofrecen referencias a patrones relacionados, que ya sea refinen al patrón actual, o manejen soluciones a problemas similares. Finalmente se otorgan créditos a las personas que ayudaron en la creación y definición de los patrones.

Con toda esta información disponible y apropiadamente distribuida en la plantilla, se debe ser capaz de entender un patrón, aplicarlo e implementarlo correctamente. La descripción del patrón considerando los puntos expuestos se muestra en la siguiente tabla:

| | |
|---|---|
| ***Nombre*** | Un nombre y un resumen del patrón. |
| ***Conocido como*** | Otros nombres del patrón. |
| ***Ejemplo*** | Un ejemplo real demostrando la existencia del problema y la necesidad del patrón. |
| ***Contexto*** | Situación en la cual el patrón puede ser aplicado. |
| ***Problema*** | El problema que el patrón maneja, incluyendo las fuerzas asociadas. |
| ***Solución*** | La solución fundamental soportando al patrón. |
| ***Estructura*** | Una especificación detallada de los aspectos estructurales del patrón, incluyendo diagramas de clases. |
| ***Dinámicas*** | Escenarios típicos describiendo el comportamiento de ejecución del patrón. Se utilizan diagramas de secuencia. |
| ***Implementación*** | Guías para implementar el patrón. |

| | |
|---|---|
| ***Ejemplo resuelto*** | Discusión de aspectos importantes resolviendo un ejemplo que no es cubierto en las secciones *Solución*, *Estructura*, *Dinámicas* e *Implementación*. |
| ***Variantes*** | Una descripción breve de variantes o especializaciones del patrón. |
| ***Usos conocidos*** | Ejemplos del uso del patrón, tomados de sistemas existentes. |
| ***Consecuencias*** | Los beneficios que el patrón provee y desventajas potenciales. |
| ***Ver también*** | Referencias a patrones que resuelven problemas similares o a patrones que refinan al patrón actual. |

# Capítulo 3

# Patrones para la Arquitectura de HIM

En este capítulo se plantea la arquitectura de diseño para HIM. La definición de la arquitectura se hace con base en patrones, lo cual facilita la descripción de la funcionalidad que se desea implementar con cierto nivel de abstracción. El modelo de la arquitectura, puede ser visto como la parte fundamental del diseño y por lo tanto debe considerar las especificaciones obtenidas en las etapas anteriores del desarrollo, y además permitir una primera aproximación a la implementación con cierto grado de generalidad.

## *Arquitectura de HIM*

En la primera versión de HIM se investigaron arquitecturas que soportaran los requerimientos, representados por las clases del análisis, así como los requerimientos no funcionales. Una primera arquitectura fue la propuesta por E++[7]. De la cual se tomó el patrón Modelo – Vista – Controlador (MVC) para la arquitectura general del sistema, así como ciertos módulos internos que sirven como guía para la implementación de las responsabilidades de los elementos del MVC.

Una vez definida la responsabilidad de cada elemento se procedió a diseñar el funcionamiento interno. Considerando esto, se han elegido patrones que permitan cumplir con las características deseadas, y en general con los puntos que un sistema para web de alta calidad debe poseer [Yang]:

- Modularidad
- Reutilización
- Extensibilidad
- Portabilidad
- Inversión del control
- Consistencia
- Escalabilidad

Incluyendo además la seguridad.

A continuación se comentan los patrones usados y la función que cumplen en la arquitectura[8].

## *Model – View – Controller*

El patrón de arquitectura MVC divide a una aplicación interactiva en tres componentes. El modelo, que contiene la funcionalidad medular y los datos. La vista, que despliega la información al usuario; y el controlador, que a partir de los datos de entrada accede al modelo y responde a las peticiones del usuario. A su

[7] Ver el documento Arquitectura del Diseño (J2EE), incluido en el CD de HIM.
[8] Se han mantenido los nombres de los patrones en inglés, debido a que la mayor parte de la literatura en español así los maneja, evitando ambigüedades en las referencias mencionadas.

vez la vista y controlador constituyen la interfaz del usuario (en algunos sistemas de patrones se conoce como módulo de presentación). Un mecanismo de propagación de cambios asegura la consistencia entre la interfaz de usuario y el modelo.

Basados en los paquetes especificados en la etapa del análisis del desarrollo de HIM[9], se estableció que entidades serían manejadas por los componentes de MVC. De este modo la interfaz humana, soportada a través de páginas web dinámicas, se incluyó en la capa de vista. La capa del dominio del problema, constituida por los paquetes de control y seguridad, se constituyó como parte del control, y finalmente la capa de manipulación de datos, que incluye el paquete de base de conocimiento formó el modelo.

El **contexto** de este patrón está claramente definido para la creación de sistemas interactivos que demandan una interfaz de usuario flexible.

El **problema** que se busca resolver considera que la interfaz de usuario está especialmente propensa a cambiar, en situaciones como:

- Cuando se extiende la funcionalidad de la aplicación, y se deben modificar los menús para acceder a las nuevas funciones.
- Un cliente puede hacer adaptaciones de la interfaz según sus necesidades.
- Se puede necesitar migrar el sistema a otra plataforma con un diferente estándar de "look & feel", o simplemente cambios en el sistema de despliegue actual (ventanas o páginas web) pueden implicar cambios de codificación.

También se debe considerar que diferentes usuarios pueden requerir distintas formas de interactuar con el sistema. Un capturista requiere ingresar información vía el teclado, mientras que un administrador quisiera usar el mismo sistema a través de íconos y botones. Consecuentemente, el soporte de varios paradigmas de interfaz de usuario debe ser fácilmente incorporado.

Construir un sistema con la flexibilidad requerida es caro y propenso a errores si la interfaz de usuario está mezclada con el núcleo funcional. Esto puede ocasionar que se necesite desarrollar y mantener distintos sistemas por cada implementación de interfaz de usuario.

Las fuerzas que influencian la solución son:

- La misma información es presentada de diferente manera en diferentes ventanas.
- El despliegue y comportamiento de la aplicación debe reflejar inmediatamente la manipulación de los datos.

[9] Ver el documento Diagramas de Clases del Análisis, incluido en el CD de HIM.

- Los cambios a la interfaz de usuario deben ser fáciles, y de ser posible en tiempo de ejecución.
- Permitir diferentes estándares de "look & feel", o migrar la interfaz de usuario no debe afectar el código en el núcleo de la aplicación.

La **estructura** para dar soporte a las características anteriores se puede distribuir como sigue:

**Modelo**

- Provee del núcleo funcional de la aplicación: la lógica y las estructuras de datos del negocio.
- Registra vistas dependientes a través de los controladores.
- Notifica a los componentes dependientes acerca de cambios en los datos. Si la información incluida en el modelo es manejada por distintos componentes, es necesario se mantenga la consistencia de los datos para lo cual se puede hacer uso de un componente sincronizador.

**Vista**

- Inicializa el controlador asociado.
- Despliega información al usuario, dependiendo del contexto.
- Implementa los procedimientos de actualización.

**Controlador**

- Acepta las entradas del usuario como eventos.
- Traduce los eventos a solicitudes de servicios del modelo, o despliega peticiones en la vista.
- Si se requiere implementa el procedimiento de actualización.
- Obtiene datos del modelo.

El siguiente diagrama (figura 3.1) representa las interacciones entre los componentes

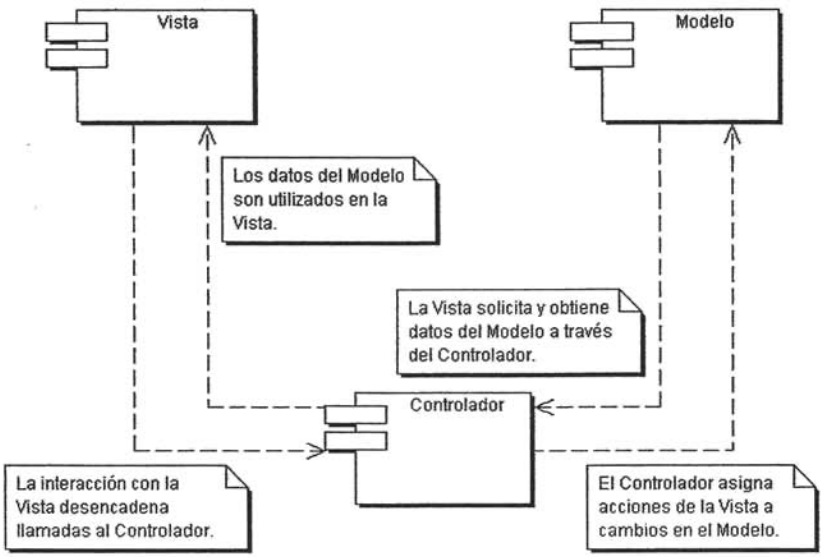



**Figura 3.1**

Una **variante** del patrón consiste en disminuir la separación de la vista y el controlador, combinando las responsabilidades de estos dos elementos en un solo componente, con lo cual se sacrifica la intercambiabilidad de los controladores, pero se disminuye el número de elementos implicados, y por consiguiente la complejidad. Esta estructura es conocida como arquitectura Documento-Vista (Document-*View*). El componente documento corresponde al modelo de MVC, y también implementa un mecanismo de propagación de cambios.

Entre los **usos conocidos** del patrón MVC están los siguientes:

- *Smalltalk*. El *framework* de interfaz de usuario del ambiente de Smalltalk utiliza MVC. Su uso incluye la construcción de componentes que son compartidos por las herramientas que conforman el ambiente de desarrollo.
- *MFC*. La variante Document-*View* es integrada en la biblioteca Microsoft Foundation Class (MFC) del ambiente Visual C++ para desarrollo de aplicaciones Microsoft.
- *Struts*. El *framework* de desarrollo de aplicaciones J2EE, *Struts* implementa el patrón MVC, en la parte de la vista y el controlador, dejando abierta la especificación del modelo.

Las **ventajas** de aplicar el MVC incluyen:

- *Múltiples vistas del mismo modelo*, sincronizadas a través del mecanismo de propagación de cambios, como el provisto por el patrón Observer[10].
- *Vistas y controladores conectables.* La separación conceptual del MVC permite el intercambio de los componentes de la vista y el controlador, aun en tiempo de ejecución.
- *Creación potencial de un framework.* Es posible establecer la base para un *framework* de aplicación con este patrón. Inicialmente este fue el objetivo con el que se desarrollo en Smalltalk y posteriormente se ha utilizado en distintas propuestas de arquitecturas J2EE.

Como otros patrones, MVC también tiene sus **desventajas**:

- *Incrementa la complejidad.* Seguir la estructura de MVC puede no ser la mejor opción en una aplicación interactiva, si ésta tiene una interfaz de usuario fija que no necesita ser modificada constantemente.
- *Número de actualizaciones excesivas.* Puede ser que no todas las vistas necesitan ser notificadas de cambios en el modelo.
- *Acoplamiento fuerte de la vista y el controlador al modelo.* Debido a que la presentación hace llamadas directas al modelo, cambios en la interfaz de este último pueden ocasionar inconsistencias en el controlador, o en la vista. Este problema consigue ser manejado aplicando el patrón *Command* Processor, o algún otro método de direccionamiento.
- *Ineficiencia en el acceso a los datos desde la vista.* Dependiendo de la interfaz del modelo, la vista puede necesitar realizar múltiples llamadas al modelo para obtener los datos a desplegar. Esto ocasiona la disminución del desempeño si los cambios son frecuentes. El almacén temporal de los datos (*cache*) puede evitar este problema.

Se da, finalmente reconocimiento a Trygve Reenskaug creador del patrón MVC, e introductor de éste en el ambiente de Smalltalk.

La arquitectura MVC no ha permanecido estancada, y aunque fiel a su diseño original a sido adaptada e implementada en distintas tecnologías y contextos (aplicaciones empresariales, sistemas standalone). En la comunidad Java se inició el proyecto *Struts* en mayo del 2000, encabezado por Craig R. McClanahan [McClanahan], con el objetivo de proveer a los programadores de Java de un *framework* estándar MVC.

[10] Ver la sección *Patrones para el Modelo* del capítulo *Patrones para la Arquitectura de HIM.*

## *Patrones para la arquitectura*

La **solución** para organizar la estructura de HIM que se propone a partir del MVC incluye el uso de distintos patrones para implementar cada uno de los componentes principales. Cabe señalar que para la especificación de estos patrones se utilizará un modelo reducido de la plantilla, incluyendo una descripción del patrón, especificación de los elementos, sus responsabilidades y cómo interactúan, y las características que confieren al implementarlos.

### *Patrones para la Vista*

No menos importante que los otros dos módulos de la arquitectura MVC, la vista es responsable de administrar la interfaz de usuario, lo cual incluye el despliegue de las pantallas apropiadas a las necesidades del usuario, el soporte a los eventos realizados por éste, y cierto manejo de la seguridad. Debido a estos puntos y a la integración de este módulo con diferentes y cambiantes tecnologías (navegadores de distintos fabricantes, sistemas operativos con bibliotecas gráficas distintas, etc.) es necesario crear un sistema de vistas, donde la lógica de la presentación esté desacoplada de la lógica del negocio. Esto hará al sistema más flexible, más reutilizable, y en general más susceptible a ser modificado.

Otro punto a analizar es que el sistema maneja peticiones a través de la Web y se ha decidido utilizar páginas web para darle soporte a la interfaz de usuario. Esto puede representar un problema debido al carácter estático de las páginas. Una manera de subsanar esta deficiencia es procesar las vistas basados en plantillas adaptables en tiempo de ejecución a peticiones específicas.

Los patrones utilizados en la vista son:

- *View Helper*
- *Composite View*
- Mediator

#### *View Helper*

La solución propuesta por este patrón delega la obtención del contenido de la vista a sus *helpers*, los cuales sirven como modelo de los datos y adaptadores de éstos. La lógica del negocio en la presentación está encapsulada en el *helper* y se sitúa entre la vista y el negocio, estableciendo un canal de comunicación entre ambas capas. Comúnmente se almacenan en un contenedor de objetos que los mantiene disponibles dependiendo de cierto alcance definido. Múltiples clientes, tales como vistas o controladores, pueden utilizar el mismo *helper* para obtener y adaptar el estado de un modelo, o para presentarlo de múltiples maneras.

La otra manera de reutilizar la lógica embebida en una vista es copiándola y pegándola en el lugar donde se necesita; lo cual hace difícil de mantener el sistema debido a la potencial propagación de errores que deben ser corregidos en múltiples lugares.

*Diagrama de clases*

A continuación se muestran las clases del patrón, figura 3.2:

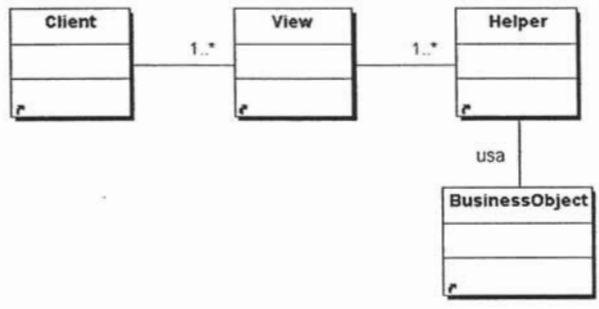



**Figura 3.2**

*Participantes y responsabilidades*

El siguiente diagrama de secuencia (figura 3.3) representa al patrón *View Helper*

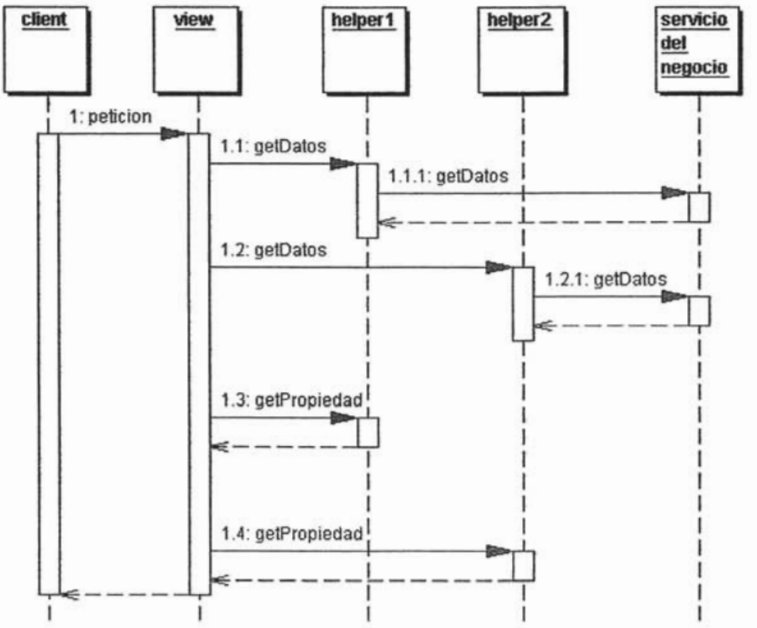



**Figura 3.3**

- *View*. Representa y muestra información al cliente; esta información es obtenida gracias a los *helpers* que encapsulan y adaptan un modelo para ser usado en el despliegue.

- *Helper*. Los *helpers* tienen como responsabilidades la reunión de los datos requeridos, y la adaptación de éstos para ser usados por una *View* (vista) o un *controller* (controlador). Los *helpers* pueden dar servicio a las solicitudes de los datos por parte de una *View*, dando acceso a datos puros, o formateados como contenido web.

Un *helper* puede representar un objeto *command*, *delegate*, o una transformación XSL, la cual es usada en combinación con una hoja de estilo para adaptar y convertir el modelo de datos en una forma apropiada.

Estas características proveen al sistema del soporte necesario para

- *Mejorar la división de la aplicación entre las distintas capas*. Usando *helpers* resulta una separación más clara de la vista y el control, al proveer un lugar para factorizar la lógica del negocio. Estableciendo esta lógica en la vista la vuelve voluminosa y difícil de manejar.
- *Mejorar la separación de roles*. Separar la lógica del formato de la lógica del negocio, reduciendo las dependencias que personas con diferentes roles

tengan sobre las mismos fuentes. Esto reduce la probabilidad de modificaciones accidentales y la introducción de errores en el sistema.

- *Permitir la reutilización*. El código no es duplicado en varias vistas, y por lo tanto mantenido y depurado. Adicionalmente, como la lógica es removida de la vista, puede reutilizarse, potencialmente sin modificación, al servicio de interfaces de usuarios totalmente distintas.

### *Composite View*

Este patrón permite combinar porciones atómicas en una vista compuesta total. Generalmente las vistas son construidas individualmente con código que define su formato directamente en ellas, ocasionando que se duplique el código y que las modificaciones al esquema sean difíciles y propensas a errores.

La solución de este patrón consta de la creación de vistas compuestas a partir de la inclusión, o sustitución, de módulos de plantillas, dinámicos o estáticos. Esto promueve la reutilización de porciones atómicas de la vista, e impulsa el diseño modular. Puede haber cierta carga de procesamiento asociada con este patrón, la cual debe ser evaluada contra la mejora obtenida en la flexibilidad.

Es común que un sistema tenga muchas vistas, algunas de las cuales cuentan con elementos comunes a ser mostrados en formas alternativas, tales como desplegarse en diferentes lugares de la pantalla, o conteniendo texto con distinto formato. Cuando un componente es codificado directamente en la vista en la cual es usado, un cambio en ese componente requiere cambiar cada vista que lo contiene, generando múltiples modificaciones para hacer un cambio simple.

*Diagrama de clases*

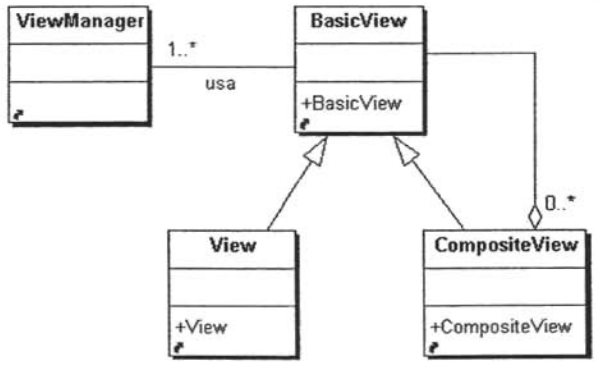



**Figura 3.4**

*Participantes y responsabilidades*

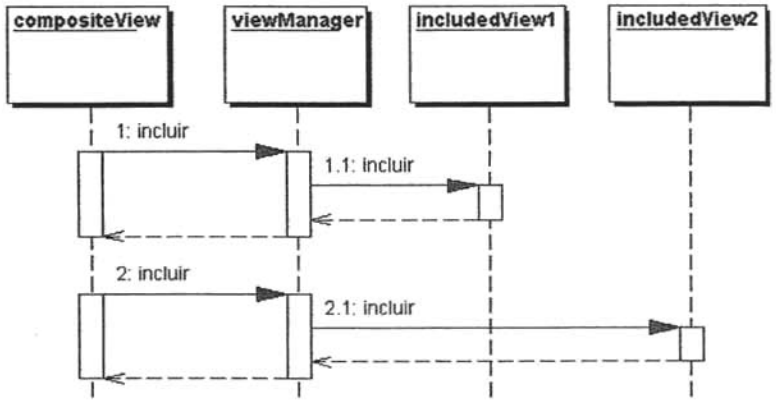



**Figura 3. 5**

- *Composite View*, es una vista formada a partir de sub-vistas.
- *View Manager*, que administra la inclusión de fragmentos de plantilla en la vista compuesta.
- *Included View* es una subvista utilizada como una pieza atómica de una vista completa más grande. Esta vista incluida puede ser a su vez una vista compuesta con múltiples subvistas.

Existen varios mecanismos, que permiten la inclusión condicional de las subvistas, por ejemplo, ciertos fragmentos de la plantilla pueden ser incluidos sólo si el usuario cumple con un rol en particular, o ciertas condiciones del sistema son satisfechas. Además, usando un componente *helper* como manejador de vistas permite un control más sofisticado de la estructura de la página como un todo, lo cual es útil para crear plantillas reutilizables.

**Mediator**

En la capa de vista se aprovecha el manejo dinámico de las pantallas proporcionado por el patrón Mediator, el cual simplifica la comunicación entre los objetos de las vista y del control, introduciendo un único objeto que gestiona la distribución de mensajes. Una variación de este patrón contempla el manejo de roles configurables, en el cual los clientes definen un rol (que podría ser cambiado conforme se ejecuta el sistema) que tendría requerimientos para los mensajes entre los componentes de la vista.

Este patrón no es implementado como tal, en su lugar se utiliza el patrón *Dispatcher* que forma parte del *Dispatcher View* (ver más adelante). Se menciona aquí únicamente para mostrar la representación de este módulo proveniente del patrón E++, en la arquitectura final del diseño.

Es importante señalar que dado que el Mediator forma parte de la conexión entre la vista y el control, se puede situar del lado del control. Esto se justifica al analizar la funcionalidad que debe implementar. Es por ello que se ha incluido al patrón *Dispatcher* en el control. Algunas especificaciones de patrones [Sun] definen una arquitectura en la cual el control y la vista están incluidas en la un mismo módulo llamado presentación, que integra patrones de vista y de control y desde este punto de vista el *Dispatcher* puede considerarse como perteneciente a ambas capas.

## *Patrones para el Control*

El control en el patrón MVC define la manera en que el sistema reaccione ante las entradas del usuario, permitiendo modificar el comportamiento del sistema sin tener que cambiar la presentación visual. Además sirve como puente de comunicación entre la vista y el modelo evitando la dependencia entre las estructuras de datos del negocio y la forma en que son presentadas en la vista.

El control también es responsable de recibir las solicitudes de los clientes, decidir que funciones de la lógica del negocio se debe llevar a cabo y finalmente delegar la responsabilidad para producir el siguiente paso en la interfaz del usuario.

Los patrones utilizados en el control son:

- *Front Controller*
- *Dispatcher View*
- *Command*

### *Front Controller*

Este patrón permite crear entre otras cosas, un punto de acceso centralizado para el manejo de peticiones, soportar la integración del sistema de servicios, obtener contenido, administrar la vista y la navegación (ver figura 3.6). Cuando el usuario accede a la vista directamente sin pasar a través de un mecanismo centralizador, pueden surgir los siguientes problemas:

- Cada vista debe proveer su propio sistema de servicios, resultando en la duplicación del código.

- La navegación de las vistas es responsabilidad de éstas, lo cual resulta en una mezcla de contenido y lógica de navegación.

El uso del *Front Controller* (FC) como punto inicial de contacto en el manejo de las peticiones, incluye la invocación de servicios de seguridad, como la autentificación y la autorización. Asimismo es responsable de delegar los procesos del negocio, es decir, una vez interpretada una petición deben invocarse los mecanismos del negocio para darle respuesta. Es encargado de elegir la vista apropiada, manejar los errores, y administrar las estrategias de creación de contenido. Estas responsabilidades son apoyadas por otros módulos basados en patrones bien definidos, como se verá más adelante en el patrón *Dispatcher View*.

El FC es configurado definiendo un mapeo de acciones (*ActionMappings*). Un mapeo de acción incluye una ruta que es relacionada entre la dirección de una solicitud, y una clase que encapsula cierta lógica para interpretar la petición y procesarla, consignando finalmente el control al componente apropiado de la vista para crear una respuesta.

*Diagrama de clases*

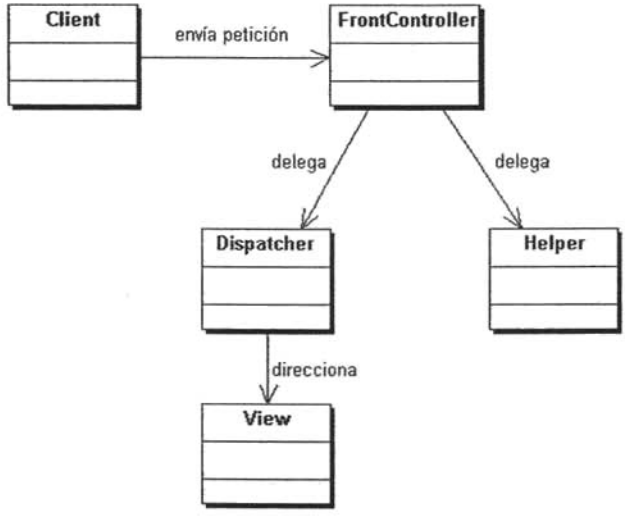



**Figura 3.6**

*Participantes y responsabilidades*

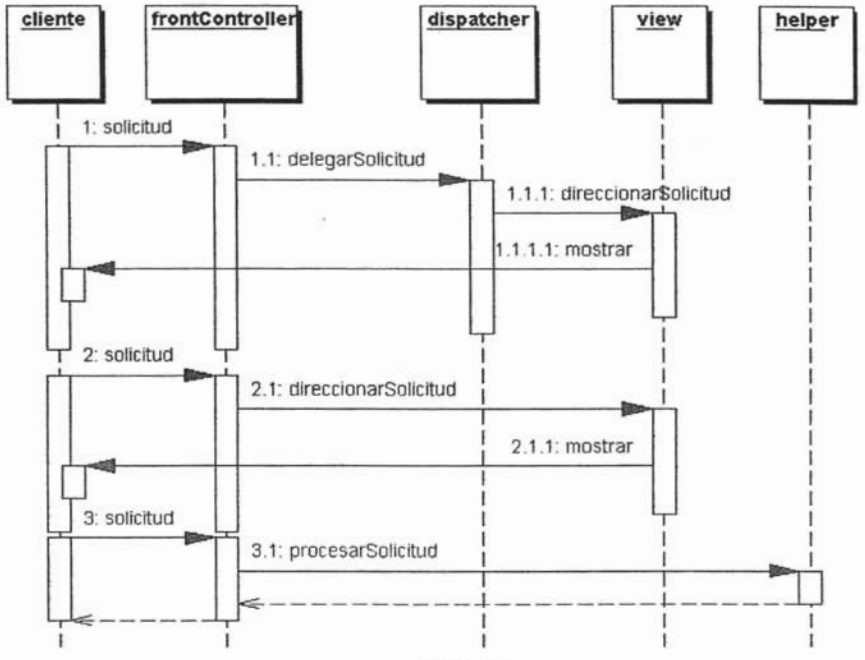



**Figura 3.7**

- *FrontController*, es el punto de contacto inicial para manejar todas las solicitudes al sistema. Generalmente interactúa con los patrones *Dispatcher*, o *Service* to Worker.

El *FrontController* puede delegar algunas tareas a un *helper*.

- *Dispatcher*, es responsable del manejo de la vista y de la navegación. Por lo tanto, el *Dispatcher* elige la vista a presentar al usuario y provee de mecanismos para el control de recursos organizados de manera vectorial. Este tipo de recursos permite incluir de manera dinámica validaciones, u otros procesos, en un vector, las cuales son ejecutadas secuencialmente y en caso de reportarse violaciones a las restricciones establecidas en alguno de los elementos del vector reenviar el control a un componente adecuado. El *Dispatcher* puede ser encapsulado en el FC, o funcionar como un componente separado.

- *Helper*, es responsable de apoyar a la *View* o al FC a llevar a cabo su procesamiento, incluyendo reunir los datos y adaptarlos para ser usada por la *View*.

- *View,* representa y despliega la información al cliente. Esta información es obtenida del modelo.

La administración de las acciones es llevada a cabo con el patrón Chain of Responsability (CR) que nos permite enlazar el objeto que manejará una petición sin conocer a priori la relación del objeto y la petición, ya que ésta es determinada dinámicamente. Comúnmente una tarea necesita la interacción de distintos objetos para ser llevada a cabo, estos objetos, representados por las acciones son concatenados y se comunican entre si gracias al patrón CR. La cadena de acciones obtenidas forma un filtro, el cual puede aplicar una serie de validaciones a las peticiones o eventos originados por los usuarios. Este mecanismo permite la asignación de las validaciones aplicadas sin tener que modificar el código existente, sólo es necesario agregar a la cadena el objeto que implemente la función requerida.

Las ventajas de usar este patrón se reflejan en los siguientes puntos:

- Debido a que ciertas decisiones, como elegir el componente al cual será reenviada una petición, deben ser hechas después de llevar a cabo cierta lógica del negocio, es importante que el proceso de tomar la decisión ocurra en un lugar centralizado, lo cual implica que la ocurrencia de errores se reduce a un solo lugar.

- Permite el control centralizado de la seguridad, y una división bien definida de la aplicación favoreciendo la reutilización, El código que es común entre componentes es colocado en el controlador y reutilizado en cada petición.

- Facilita el seguimiento en el manejo de la aplicación por parte de un usuario particular, registrando el historial de las actividades realizadas. También maneja la validación y los errores, ya que estas operaciones son hechas frecuentemente en las peticiones.

### *Dispatcher View*

Este patrón combina elementos de los patrones *Dispatcher* y *View Helper* (ver figura 3.8), para manejar las solicitudes de los clientes y preparar una presentación dinámica como respuesta. El *Dispatcher* es responsable de la administración de la vista y la navegación, y puede ser encapsulado dentro de un controlador (*Front Controller*), o trabajar como un componente separado, pero en coordinación, ver figura 3.9.

Este patrón describe una estructura similar a la propuesta en el patrón *Service* to Worker (S2W), con diferencias en la división del trabajo entre los componentes. El *Controller* y el *Dispatcher* tienen responsabilidades más limitadas comparadas con el S2W, ya que el procesamiento *front*al y la lógica para la administración de las vistas son muy básicos. Más aún, si es considerado innecesario un control centralizado de los recursos internos, el *Controller* puede ser removido y el *Dispatcher* trasladado a la capa de vista.

En el patrón *Dispatcher View* (DV), el *Dispatcher* juega un rol de limitado a moderado en la administración de las vistas; mientras que en el patrón S2W su papel es de moderado a extensivo. Un rol limitado para el *Dispatcher* se da cuando no son utilizados recursos externos para elegir una vista. La información encapsulada en la petición es suficiente para determinar la vista y responder a la solicitud. Un ejemplo teniendo un rol moderado es en el caso de un cliente enviando una solicitud al *Controller*, con un parámetro de consulta que describe una acción que debe ser llevada a cabo. En este caso el *Dispatcher* debe procesar la solicitud y decidir si son necesarios recursos extras para responderla.

*Diagrama de clases*

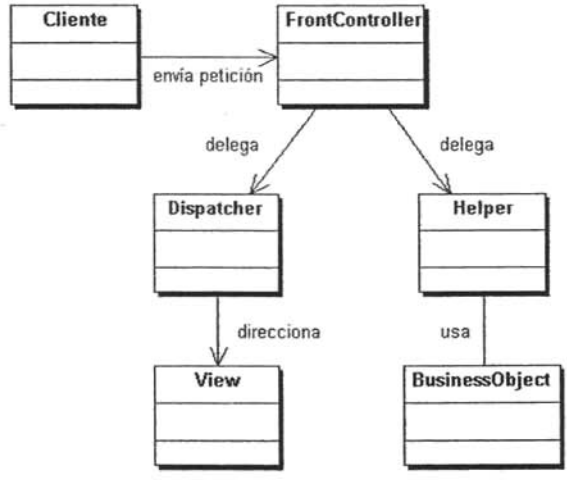



**Figura 3.8**

*Participantes y responsabilidades*

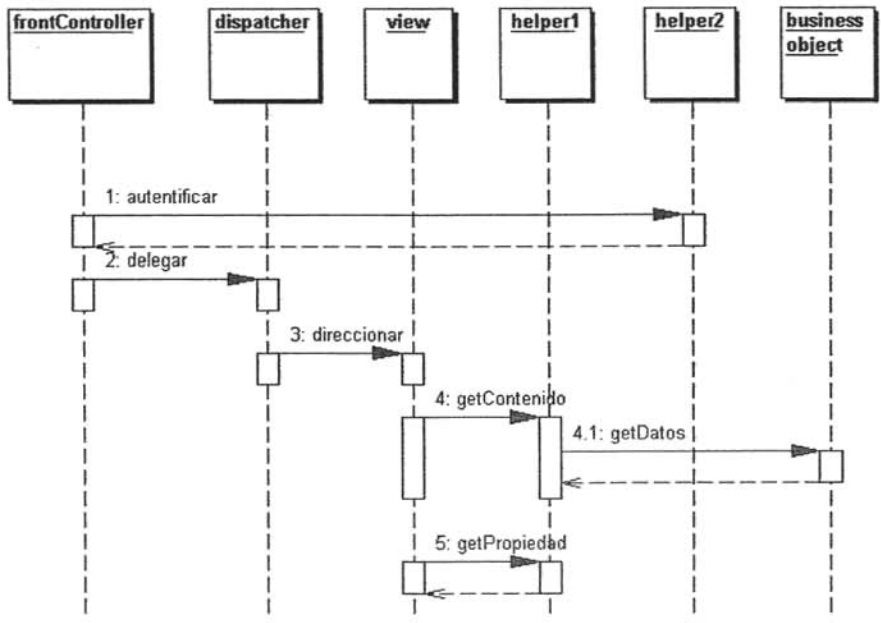



**Figura 3.9**

- *FrontController*, es el punto de contacto inicial para manejar las solicitudes al sistema. Maneja la autentificación y delega a un *Dispatcher* la administración de la vista.

- *Dispatcher*, es responsable del manejo de la vista y de la navegación. Por lo tanto, el *Dispatcher* elige la vista a presentar al usuario y provee de mecanismos para el control de este recurso. El *Dispatcher* puede ser encapsulado en el FC, o funcionar como un componente separado.

- *Helper*, es responsable de apoyar a la *View* o al FC a llevar a cabo su procesamiento, incluyendo reunir los datos y adaptarlos para ser usados por la *View*.

- *View*, representa y despliega la información al cliente. Esta información es obtenida del modelo.

- *BusinessService*, representa funcionalidad propia de la aplicación, que se encuentra disponible a los clientes.

Las ventajas de usar este patrón se reflejan en los siguientes puntos:

- El control centralizado provee de un lugar para manejar los servicios del sistema a través de múltiples peticiones, aunque algunas decisiones pueden estar incluidas como parte de la petición, el control centralizado permite un mejor manejo de los servicios de seguridad y de reenvío.

- El código es mantenido en un solo lugar en vez de duplicarlo en cada una de las vistas y por lo tanto, es más fácil de mantener y depurar. Adicionalmente, el remover la lógica de la vista permite la reutilización de la misma lógica del negocio, potencialmente sin modificación, para servir a interfaces de usuario completamente distintas a las páginas web, como los sistemas de manejo de ventanas de algunos sistemas operativos.

### *Command*

Gracias a este patrón es posible crear los objetos responsables de manejar las solicitudes (acciones, ver *Front Controller*) y respuestas a los clientes, o indicar que el control debe ser reenviado a algún otro módulo del sistema, desacoplando la fuente de una petición del objeto que la cumple, ver figuras 3.10 y 3.11. Por ejemplo, si es invocada la función de entrada al sistema (*login*), un objeto *command* deberá implementar la lógica para procesar la solicitud: requerir datos a la capa del modelo, aplicar algún método de validación, y si ésta es exitosa reenviar la solicitud a la capa de vista para producir una interfaz de bienvenida al sistema.

*Diagrama de clases*

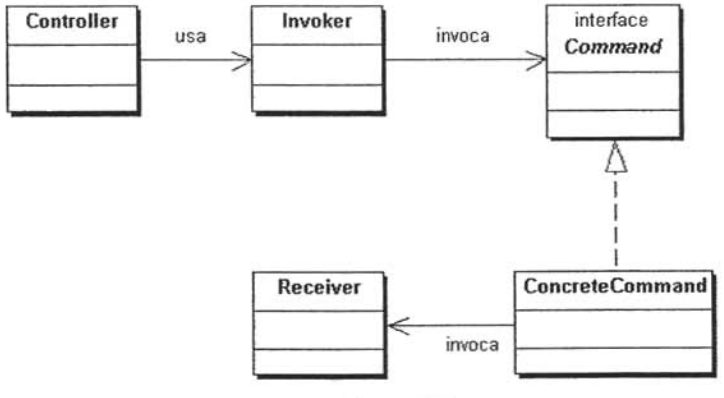



**Figura 3.10**

*Participantes y responsabilidades*

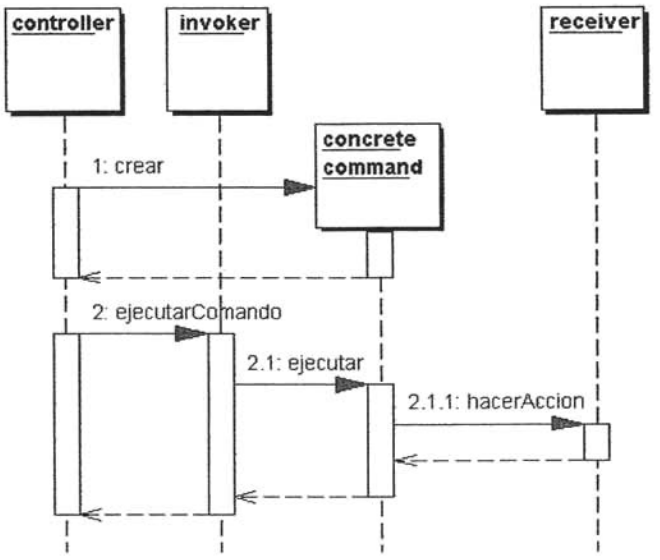



**Figura 3.11**

- *Command*, en esta interfaz se definen los métodos utilizados por el Invoker.
- *Invoker,* invoca el método ejecutar() del objeto *Command*.
- *Receiver,* es el objeto que realmente lleva a cabo la tarea solicitada.
- *ConcreteCommand,* implementa la interfaz *Command*. Mantiene una referencia al Receiver. Cuando se llama al método ejecutar(), Concrete*Command* llama a uno, o más métodos del Receiver.
- *Controller,* puede ser cualquier elemento que requiera realizar una tarea, por ejemplo un *Front Controller*.

Entre las ventajas provistas por este patrón está el encapsular un comando en un objeto de forma que puede ser almacenado, pasado como parámetro o devuelto por un método, es decir, cuenta con todas las propiedades de un objetos.

Estas característica proveen al sistema del soporte necesario para

- *Deshacer comandos*. La ejecución de operaciones con objetos *Command* pueden almacenar su estado para ser revertidas por si mismas. La interfaz del *Command* debe tener un método *deshacer* que revierta los efectos de una llamada previa al método *hacer*. Los comandos ejecutados son almacenados

en un historial, el cual permite deshacer o rehacer según el sentido en que se recorra la lista, y redireccionando las llamadas a los métodos apropiados de los objetos *Command*.

- *Tener procesos de identificación.* Soporta el registro de un historial, de manera que un proceso puede ser ejecutado nuevamente en caso que el sistema falle. Extendiendo la interfaz de *Command* con operaciones de *carga* y *almacenamiento*, es posible mantener persistencia en el historial.

- *Manejar transacciones.* Permite estructurar el sistema alrededor de operaciones de alto nivel construidas sobre operaciones primitivas, ofreciendo una manera de modelar transacciones. Los objetos *Command* comparten una interfaz común, permitiendo invocar las transacciones de la misma manera.

- *Establecer colas de comandos.* Un objeto *Command* puede tener un tiempo de vida independiente de una petición. Si el receptor de la petición puede ser representado en un espacio de nombres de manera independiente (*actionMappings*, ver *Front Controller*), es posible transferir el objeto *Command*, a otro proceso y terminar la petición posteriormente.

## *Patrones para el Modelo*

Por su parte, la capa del modelo se encarga de la persistencia de los datos, la consistencia de éstos y la creación de los objetos del negocio. El estado actual del sistema es representado por un conjunto de objetos (ver *Value Objects* más adelante), cuyas propiedades definen el estado que el sistema debe presentar a un usuario para una petición específica. Adicionalmente el modelo debe manejar la conexión con aplicaciones de terceros, como sistemas heredados, o bases de datos distribuidas, de manera transparente para el resto de la aplicación.

Los patrones utilizados en el modelo son:

- *Data Access Object*
- *Observer*
- *Session Façade*

### *Data Access Object*

Este patrón permite abstraer y encapsular el acceso a las fuentes de datos. Los *Data Access Object* (DAO) implementan los mecanismos de acceso requeridos para trabajar con las fuentes de datos, sin importar de donde provengan (ver figura 3.12). Los componentes del negocio dependen de la interfaz que los DAO exponen a sus clientes, ocultando completamente los detalles de la lógica de acceso a los datos. Debido a que la interfaz usada por los clientes no

cambia aunque la implementación si lo haga, el patrón permite adaptar los DAO a diferentes esquemas de almacenamiento sin afectar a sus clientes ni a los componentes del negocio. Esencialmente el DAO actúa como un adaptador entre el componente y la fuente de datos.

*Diagrama de clases*

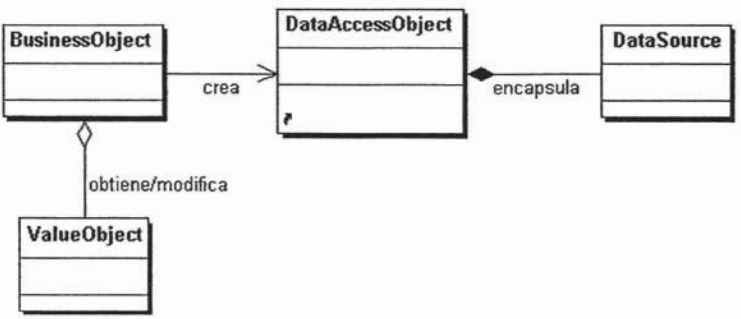



**Figura 3.12**

*Participantes y responsabilidades*

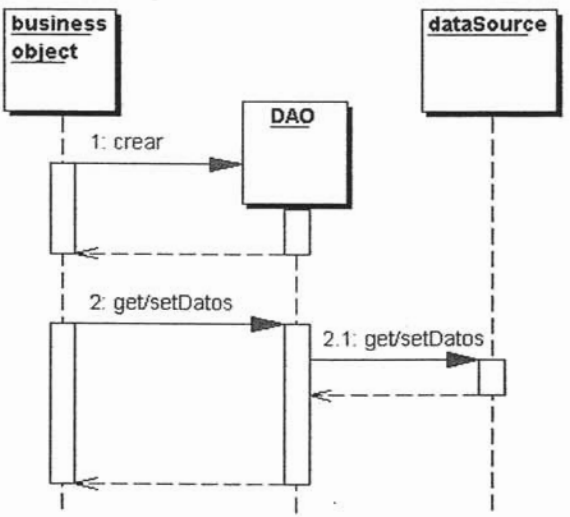



**Figura 3.13**

- *BusinessObject*, representa al cliente y es el objeto que requiere el acceso a la fuente de datos.

- *DataAccessObject*, abstrae de la implementación subyacente de acceso a los datos, para permitir un manejo transparente de las fuentes de datos.

- *DataSource*, representa la implementación de una fuente de datos, como puede ser un RDBMS, OODBMS, un repositorio XML, un sistema de archivo plano. Una fuente de datos puede también ser otro sistema, un servicio, o algún tipo de repositorio (LDAP).

En HIM la persistencia es soportada con RDF – XML, a través de bibliotecas de Jena, y para la traducción de los datos desde XML a objetos se usa el patrón DAO [Hernández].

Las ventajas de usar este patrón se reflejan en los siguientes puntos:

- *Permite la transparencia.* Los objetos del negocio pueden usar las fuentes de datos sin conocer los detalles de implementación específicos de éstas.

- *Facilita la migración entre diferentes implementación de persistencia.* La migración implica cambios únicamente en la capa de los DAO. Más aún, si se emplea una estrategia de fábrica, es posible proveer una implementación concreta de fábrica para cada sistema de almacenamiento; en este caso la migración a una persistencia diferente se reduce a proveer de una nueva implementación de fábrica a la aplicación.

- *Reduce la complejidad en la codificación en los objetos del negocio.* Debido a que el DAO maneja toda la lógica de acceso a los datos, esto simplifica el código de los objetos del negocio, mejorando la legibilidad del código y la productividad del desarrollo.

- *Centraliza el acceso a los datos en una capa separada.* Debido a que las operaciones de acceso a los datos son delegadas a los DAO, la capa formada por estos puede verse como la capa que aísla al resto de la aplicación de la implementación de la persistencia, haciendo más fácil de mantener y administrar la aplicación.

### *Observer*

Una responsabilidad que debe ser revisada al particionar un sistema en una colección de clases cooperativas (clases encargadas de llevar a cabo subtareas de una tarea mayor) es mantener la consistencia en los objetos relacionados. Sin embargo al considerar esto hay que tener cuidado en no incrementar el acoplamiento entre las clases, ya que esto reduciría la reutilización.

En HIM se encuentra el caso anterior, debido a que el sistema requiere manejar información interrelacionada a distintos niveles: documentos, el modelo del negocio, meta información; y cada uno de estos niveles están soportados de distinta manera, es necesario contar con un mecanismo que garantice la consistencia de los datos.

Para lograr esto se utilizó el patrón *Observer* que permite tener un medio centralizado que capte las modificaciones en las capas de datos y reporte los cambios a las partes afectadas para una actualización apropiada (ver figura 3.15). Este módulo se nombró Sincronizador, el cual debe ser único para garantizar la consistencia, es decir, como objeto del sistema sólo puede instanciarse una vez, para evitar la duplicación de mensajes. Este comportamiento de unicidad se logra con el patrón *Singleton*.

*Diagrama de clases*

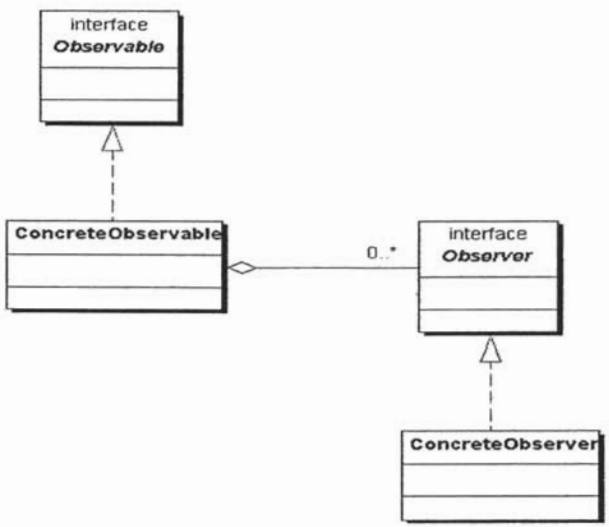



**Figura 3.14**

*Participantes y responsabilidades*

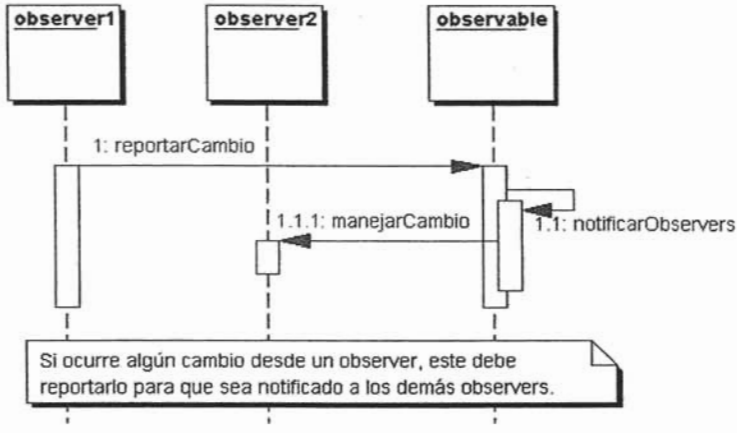



**Figura 3.15**

- *Observable*, en esta interfaz se define cómo pueden interactuar los observadores con un Observable. Estos métodos deben incluir la introducción, eliminación y la notificación de *Observers*.

- *ConcreteObservable,* clase que proporciona implementaciones para todos los métodos de la interfaz Observable. Necesita mantener una colección de *Observers*.

- *Observer,* interfaz utilizada por los observadores para comunicarse con el objeto Observable.

- *ConcreteObserver,* implementa la interfaz *Observer*, y determina cómo responder a los mensajes recibidos de Observable.

Las ventajas de usar este patrón se reflejan en los siguientes puntos:

- *Cuando una abstracción tiene variados aspectos dependientes unos de otros*, como es la separación de las fuentes de los datos dependiendo de la información manejada. Encapsular estos aspectos en objetos separados permite variar y reutilizarlos independientemente. Esto facilita la inserción de nuevos sistemas de manejo de datos.

- *Cuando un cambio en un objeto requiere realizar cambios en otros*, y no se sabe a priori cuantos objetos deben ser cambiados. En el caso del repositorio y el marco de trabajo, pueden estar contenidos en una sola clase cada uno o estar distribuidos en distintos módulos, o en diferentes computadoras; pero

desde el punto vista de la información son manejados como entidades individuales que deben ser notificadas y actualizadas consistentemente.

- *Permite a un objeto notificar a otros objetos sin hacer suposiciones acerca de ellos.* En otras palabras, no es necesario que los objetos relacionados estén fuertemente acoplados. Esto se logra con la definición de la interfaz para los objetos observados. Debido a que todos los objetos que deben ser notificados implementan esta interfaz pueden ser tratados de manera uniforme por el *Observer* sin necesidad de conocer la implementación de cada uno de ellos.

### *Session Façade*

Usando este patrón se encapsulan las interacciones entre los objetos del negocio, ver figura 3.16. El patrón *Session Façade* (SF) provee una sola clase para la interacción entre el control y el modelo. El módulo que realiza esta función recibe peticiones sobre los objetos del negocio y devuelve objetos con valores (*Value Objects*), los cuales son un reflejo de los objetos del negocio pero únicamente contienen datos y métodos de acceso (lectura / escritura).

La SF abstrae las interacciones de los objetos del negocio subyacentes en el control y provee de una capa de servicios que expone únicamente las interfaces requeridas (figura 3.17). Por lo tanto la SF maneja las relaciones entre los objetos del negocio, administrando sus ciclos de vida: creando, localizando, modificando y destruyendo los objetos requeridos en el flujo de trabajo. En aplicaciones complejas la SF puede delegar la administración de los ciclos de vida a un módulo separado, por ejemplo a un *Service* Locator.

Es importante examinar las relaciones entre los objetos del negocio. Algunas pueden ser transitorias, lo cual significa que la relación es aplicable en cierto escenario. Otras son más permanentes. Las relaciones transitorias son mejor modeladas como flujos de trabajo en la SF; mientras que las permanentes entre dos objetos deben ser estudiadas para determinar qué objeto (si no es que ambos) debe mantener la relación.

Diagrama de clases

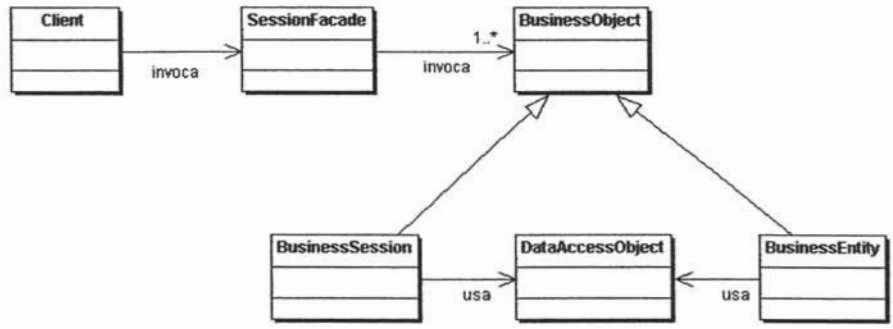



**Figura 3.16**

*Participantes y responsabilidades*

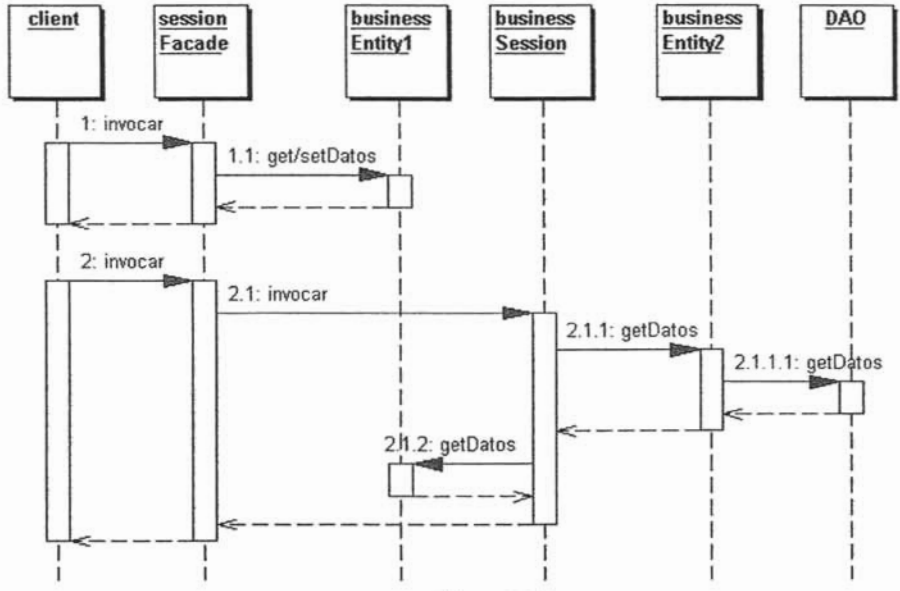



**Figura 3.17**

- *SessionFacade,* maneja las relaciones entre numerosos *BusinessObjects*, y provee de un alto nivel de abstracción al cliente. La SessionFacade ofrece una granuralidad gruesa en el acceso a los recursos subyacentes.

- *BusinessObjects,* proveen de datos y/o servicios.
- *Client,* cualquier elemento que requiere el servicio de un componente.

Las ventajas de usar este patrón se reflejan en los siguientes puntos:

- *Introduce una capa de frontera para el control.* La SF puede representar una *front*era entre los clientes y la capa del negocio, facilitando el mantenimiento de cada capa de la aplicación.
- *Establece una interfaz simple y uniforme.* Abstrae la complejidad entre los componentes del negocio participando en un caso de uso y presenta a los clientes una interfaz que es fácil de entender y utilizar.
- *Reduce el acoplamiento e incrementa la manejabilidad.* Usando una SF se desacoplan los objetos del negocio de los clientes. Se recomienda usar una SF para manejar el flujo de trabajo entre objetos, en lugar de relacionarlos directamente unos de otros. Un objeto del negocio debe ser únicamente responsable de si mismo (su estado). Esto provee una mejor manejabilidad, centralizando las interacciones, mayor flexibilidad y habilidad para enfrentar cambios.
- *Disminuye el impacto en el desempeño.* La SF reduce la sobrecarga de la red, o del sistema, entre los clientes y el servidor debido a que su uso elimina la interacción directa entre los clientes y los objetos y servicios del negocio, en su lugar todas las interacciones son canalizadas a través de la SF.

# Capítulo 4

# Tecnologías y Estrategias para Patrones

En este capítulo se proponen distintas tecnologías y estrategias para apoyar la implementación de los patrones descritos en el capítulo anterior. Esto permitirá elegir de manera objetiva el esquema que mejor se adapte para diseñar HIM. Cabe señalar que algunas de las estrategias se complementan, es por ello que no deben ser leídas de manera aislada. Las estrategias están organizadas en correspondencia con los módulos del patrón MVC, y dentro de cada uno de ellos con los patrones especificados anteriormente.

## *Tecnologías y Estrategias para la Vista*

En el capítulo anterior se explicaron los patrones que son utilizados en la vista:

- *View Helper*
- *Composite View*

### *Patrón View Helper*

La vista puede trabajar con un número indeterminado de *helpers*[11], los cuales son generalmente implementados como *JavaBeans* y etiquetas personalizadas (*JSP* 1.1+). Adicionalmente, un *helper* puede representar a un objeto *command*, a un *delegate*, o a una transformación XSL, la cual es usada en combinación con una hoja de estilo para adaptar y convertir el modelo en la forma apropiada.

**Estrategias**
***JSP View***

Esta estrategia sugiere el uso del *JSP* como componente de la vista. Aunque semánticamente es equivalente a la estrategia *Servlet View*, es una solución más elegante y por lo tanto más usada. Consiste en incrustar elementos dinámicos, como *helpers*, etiquetas *JSP* o bloques de código Java, en una plantilla estática, generalmente HTML. Posteriormente es procesada para producir la pantalla que es mostrada al usuario. La ventaja de esta estrategia consiste en la separación bien delimitada entre los elementos estáticos y dinámicos, permitiendo a un diseñador encargarse de los primeros para que posteriormente un programador añada el contenido dinámico.

[11] Objetos encargados de realizar tareas específicas que generalmente complementan una tarea más general, ver patrón *View Helper* en el capítulo anterior.

***Servlet View***

Se basa en incrustar las etiquetas de marcado directamente en el código Java; esto puede hacer más difícil la actualización y modificación del esquema de la vista. Otra consecuencia de esta estrategia es que muchas personas (los equipos de desarrollo de software y de producción web) comparten un mismo recurso, lo cual reduce la eficiencia en el ambiente de trabajo e incrementa la complejidad en el control de los recursos.

***JavaBean Helper***

Usando *helpers* se obtiene una separación más clara de la vista y los procesos del negocio. En este caso la lógica del negocio es encapsulada en un *JavaBean*, el cual ayuda en la obtención del contenido y adapta y almacena el modelo para ser usado por la vista.

***Custom Tag Helper***

La aplicación de esta estrategia requiere más trabajo *front*al que el necesario en la estrategia anterior (*JavaBean Helper*) debido a que el desarrollo de los *Custom Tags* (etiquetas personalizadas) es relativamente más complicado, y la complejidad en la integración y administración es también mayor, ya que incluye numerosos artefactos que deben ser configurados, como las etiquetas en si, la biblioteca descriptora y archivos de configuración.

Los *Custom tags helpers* pueden cumplir con los roles atribuidos a los *JavaBeans helpers*, excepto por el comportamiento de objeto *command*. En cambio los *Custom tags helpers* son más convenientes para el control de flujo e iteraciones en la vista; permitiendo encapsular la lógica, que de otra forma, estaría incrustada directamente en el *JSP*, como un código scriplet. Otra área donde los *Custom tags* son preferidos es al formatear datos en bruto para ser mostrados. Un *Custom tag* es capaz de iterar sobre una colección de datos, formatearlos en una tabla HTML, e incrustar la tabla en una vista *JSP* sin requerir código scriplet.

***BusinessDelegate Helper***

Los componentes *helpers* frecuentemente hacen invocaciones a la capa del negocio. Se sugiere el uso de *BusinessDelegate* para ocultar los detalles del manejo de la solicitud. Tanto el *Helper* como el *BusinessDelegate* pueden ser implementados como *JavaBeans*. Esta noción combinada de los dos componentes ve al *BusinessDelegate* como una especialización del *Helper*.

***Transformer Helper***

El *helper* es implementado como una *eXtensible Stylesheet Language Transformer* (XSLT, hoja de transformación de estilo extendible). Esto es particularmente útil cuando el modelo existe como un documento de marcado

estructurado, tal como XML. Esta estrategia ayuda a reforzar la separación del modelo y la vista, debido a que mucho del marcado debe ser factorizado en una hoja de estilo separada.

## *Patrón Composite View*

Como su nombre lo indica, una vista compuesta (*Composite View*) es una vista formada a partir de múltiples subvistas.

El componente que provee la funcionalidad necesaria para manipular la inclusión de los fragmentos de vistas, y por lo tanto la creación de una vista completa es el *View Manager*. Este componente puede ser implementado como un motor de tiempo de ejecución *JSP*, en la forma de la etiqueta "*include*" (*<jsp:include>*), o encapsulada en un *JavaBean helper*, o como *Custom tags*; adicionalmente puede ser implementado como una XSLT.

### *Estrategias*

Algunas de las estrategias aplicadas a este patrón han sido definidas anteriormente y sólo se mencionan para ser consideradas: *JSP View* y *Servlet View* tienen las mismas características que las descritas en el patrón *View Helper*. Cualquier ventaja, o desventaja, que puedan tener al ser aplicadas en este patrón es similar al anterior.

### *JavaBean View Manager*

La vista usa *JavaBeans*, para implementar la lógica que controla las decisiones del *view manager* relacionadas con los elementos que formarán la página. Estas decisiones pueden estar basadas en roles o políticas, logrando una funcionalidad mucho más poderosa que el estándar *Servlet / JSP*. Aunque esta estrategia es semánticamente equivalente a la *Custom tag View Manager*, no es tan elegante debido a que introduce código scriplet en la vista.

El uso de la estrategia *JavaBean* requiere menos trabajo *front*al que el desarrollado en un *custom tag*, debido a que es más fácil construir un *JavaBean* e integrarlo en un ambiente *JSP*. Adicionalmente se facilita la administración al ser el *JavaBean* es el único artefacto que hay que operar y configurar.

### *Custom Tag View Manager*

La lógica implementada en un *tag* controla las decisiones del *view manager* relacionadas con los elementos que formarán la página. Estas etiquetas extienden el comportamiento del *JSP* estándar, incluyendo acciones más poderosas. Estas acciones son capaces de tomar decisiones sobre la construcción de la página basadas en roles de usuarios o políticas de seguridad.

Como se comentó en la estrategia anterior, el uso de *custom tag* requiere más trabajo *front*al que los *JavaBeans*, esto incluye el desarrollo, la administración e integración de las etiquetas. Además esta estrategia requiere numerosos artefactos, como son la etiqueta, una biblioteca descriptora, archivos de configuración y la adecuación del ambiente para estos artefactos.

***Transformer View Manager***

Cuando se combinan varias estrategias de administración de la vista, la página puede ser construida desde numerosas fragmentos de plantilla. Estos fragmentos pueden estar basados en hojas de estilo, también compuestas, y de esta manera obtener hojas de estilo globales que importan e incluyen hojas de estilo más pequeñas.

***Early-Binding Static Resource***

Esta estrategia, como su nombre lo indica, establece el encadenamiento de los recursos en tiempo de traducción, como puede lograrse con la directiva "*include*" de *JSP* estándar. Esto provee de flexibilidad en el mantenimiento y la actualización de plantillas relativamente estáticas. Su uso es recomendado si la vista incluye encabezados y pies de página que cambian poco.

***Late-Binding Static Resource***

La premisa de esta estrategia es incluir las partes de una plantilla en tiempo de ejecución, como puede lograrse con la acción "*include*" de *JSP* estándar.

Esta estrategia puede verse como una alternativa a la anterior, cuando aspectos de la vista cambian frecuentemente, como incluir un encabezado en tiempo de ejecución que permita reflejar los últimos cambios en la vista. Más aún la inclusión dinámica puede estar basada en políticas o criterios, tales como el rol del usuario o listas de control de acceso.

***Late-Binding Dynamic Resource***

Esta estrategia es semánticamente equivalente a aplicar recursivamente el patrón *Composite View*, debido a que los recursos dinámicos incluidos como fragmentos de una vista particular pueden a su vez ser vistas compuestas. Pueden surgir inconvenientes relacionados con la eficiencia debido costos de ejecución por la inclusión dinámica.

## *Tecnologías y Estrategias para el Control*

Los patrones utilizados en el control son:

- *Front Controller*
- *Dispatcher View*
- *Command*

### *Patrón Front Controller*

El *Controller* provee un punto de entrada centralizado para la administración de las peticiones web, esto permite reducir la cantidad de código Java embebido en los *JSPs* en forma de *scriplets*.

#### *Estrategias*

#### *ServletFront*

Se encarga de manejar el procesamiento de las solicitudes, incluyendo aspectos de administración y control. Debido a que estas responsabilidades son lógicamente independientes de la vista, es más apropiado incluirlas en un *Servlet* en lugar de un *JSP*.

La aplicación de esta estrategia tiene algunas desventajas, en particular no aprovecha ciertas características del entorno de ejecución de los *JSPs*, como es la asignación automática de los parámetros de una solicitud a un *helper*. Afortunadamente estos inconvenientes son mínimos ya que es relativamente fácil crear u obtener utilerías similares, y además está la posibilidad de que algunas funcionalidades de los *JSP* sean incluidas como características estándares de los *Servlet*s en futuras especificaciones.

#### *JSPFront*

Como se comentó en la estrategia anterior el uso de *JSP* para el manejo de las solicitudes no es recomendado, ya que no está relacionado con el formato de la vista. Otro punto en contra es que el desarrollo de esta estrategia requiere que el desarrollador trabaje con el marcado de la página para poder modificar la lógica en el manejo de las peticiones, haciendo más difícil completar el ciclo de desarrollo: codificación, compilación, prueba y depuración.

#### *Command* y *Controller*

Basada en el patrón *Command*, esta estrategia sugiere proveer una interfaz genérica de los componentes *helpers*, con la cual el *controller* pueda delegar

responsabilidades, minimizando el acoplamiento entre los componentes (ver el patrón *ViewHelper*). De esta manera modificar el trabajo que debe ser llevado a cabo por los *helpers* no requiere ningún cambio en la interfaz; proveyendo de flexibilidad y de mecanismos para una fácil extensibilidad en el manejo de las peticiones.

Finalmente, como el procesamiento de los comandos no está acoplado con la invocación de los mismos desde la vista, los mecanismos de procesamiento pueden ser reutilizados por distintos clientes, no únicamente navegadores web.

***Logical Resource Mapping***

Las peticiones son enlazadas a nombres lógicos en lugar de especificar recursos físicos. Las referencias entre los nombres pueden ser especificadas y modificadas de manera declarativa en archivos de configuración.

***Multiplexed Resource Mapping***

Esta es una subestrategia de la anterior, en la que no sólo se mapea un nombre lógico a un recurso físico, sino que un conjunto completo de nombres lógicos son asignados a un mismo recurso. Esta estrategia es usada por el motor de *JSP* para asegurar que las solicitudes de recursos *JSP* son procesados por manejadores específicos.

## *Patrón Dispatcher View*

Este patrón combina un *dispatcher* con *Views* y *helpers* para manejar las peticiones del cliente y prepara una presentación dinámica como respuesta. El *dispatcher* es responsable de la administración de la vista y la navegación, y puede ser encapsulado en el *controller*, o trabajar como un componente separado en coordinación.

***Estrategias***

Algunas de las estrategias aplicadas a este patrón han sido definidas anteriormente por lo que sólo se mencionan para ser consideradas: *JSP View*, *Servlet View*, *JavaBean Helper* y *Custom Tag Helper* tienen las mismas características que las descritas en el patrón *View Helper*; por su parte las estrategias *ServletFront* y *JSPFront* comparten características con las correspondientes del patrón *Front Controller*. Cualquier ventaja, o desventaja, que puedan tener al ser aplicadas en este patrón es similar a los anteriores.

### *Dispatcher* en *Controller*

Dado el limitado rol del *dispatcher*, es común incluirlo en el *controller*. En este enfoque el *controller* no crea explícitamente un objeto *dispatcher*, en su lugar se hace cargo de reenviar la petición a la vista.

### *Dispatcher* en *View*

Si el *controller* es removido debido a su rol limitado, el *dispatcher* puede ser incluido en la vista. Este diseño es útil en casos donde sólo se tiene una vista que es mapeada a una petición específica, y una vista secundaria puede ser usada bajo casos infrecuentes. Un ejemplo de esta situación se tiene cuando un usuario no ha sido autentificado, y solicita acceso a un *JSP* protegido. Debido a que el sitio tiene pocas páginas protegidas y un contenido dinámico limitado, la autentificación puede llevarse a cabo dentro de esos *JSPs*, en lugar de usar un *controller* centralizado. Las páginas que requieren este servicio pueden incluir una etiqueta personalizada *helper* al principio de la página. Este *helper* lleva a cabo el servicio, y se encarga de desplegar la página solicitada, o de reenviar la página de captura.

### *Transformer Helper*

Como ya se ha comentado esta estrategia es especialmente útil cuando el modelo consiste de recursos de marcado estructurado, tales como XML.

El punto inicial del modelo es el *controller* que maneja las peticiones e invoca objetos *command*, implementados como *JavaBeans Helpers*. El objeto *command* inicia la obtención de datos, invocando servicios del negocio. Los datos son encapsulados en forma de *value objects* e implementados como *JavaBeans*.

Una vez que la obtención del contenido se completa, el control es dirigido a la vista adecuada, la cual usa una etiqueta de transformación para manipular el estado del modelo. La transformación se basa en una hoja de estilo, que describe cómo formatear el modelo y es obtenida comúnmente como un archivo estático, aunque también puede ser generada dinámicamente.

## *Patrón Command*

Un objeto *command* se encarga de llevar a cabo tareas que representan la lógica del negocio. Esta lógica puede estar directamente implementada en el *command* o ser solicitada por éste a los servicios del negocio. Otra responsabilidad de los objetos *command* es obtener los datos necesarios para responder una petición, generalmente esto se logra con la creación de *JavaBeans* que representan parte del modelo y son responsables de conseguir los datos que contienen.

Dada las características del patrón *Command*, entre ellas, aislar las acciones de los usuarios en objetos, se puede especificar el estado completo del sistema en tiempo de ejecución, a partir de estos objetos. La arquitectura E++ establece un componente *state machine* que hace uso del patrón *Command* para garantizar que el sistema mantiene estados válidos.

#### *Estrategias*

#### *JavaBean*

El enfoque más apropiado para implementar el patrón *Command* es a través de *JavaBeans*. Debido a mecanismos como la introspección es posible asignar parámetros automáticamente desde las vistas para ser procesados por objetos *command*. Si la petición o tarea a realizar no contiene parámetros es posible desarrollar los *command* como objetos Java nativos.

Se considera que las estrategias basadas en *JSP* no son convenientes para este tipo de objeto, ya que el mezclar la lógica del negocio dentro de la vista provocaría un acoplamiento fuerte y una definición imprecisa de los componentes de vista y control. Por otro lado la implementación de objetos *command* como *Servlet*s tampoco es adecuada, ya que los *command* no deben ser responsables de recibir peticiones web, uso básico de los *Servlet*s. Además dada la estructura de *command* debe ser posible sustituir cualquiera de ellos sin ocasionar cambios en el resto del sistema, lo cual se facilita entre menos interfaces u clases deba extender el objeto.

## *Tecnologías y Estrategias para el Modelo*

Los patrones utilizados en el modelo son:

- *Data Access Object*
- *Observer*
- *Session Façade*
- *Business Objects*
- *Value Object*

### *Patrón Data Access Object*

Este patrón permite abstraer y encapsular el acceso a las fuentes de datos, administrando la conexión al módulo de persistencia.

### *Estrategias*

#### Herramientas de generación automática de código DAO

Dado que cada objeto del negocio corresponde a un objeto DAO específico, es posible establecer la relación entre los dos elementos y la implementación subyacente (tales como tablas en un base de datos relacional o RDBMS). Una vez que las relaciones son constituidas se crea una utilería específica para la aplicación que genere los DAOs. Los meta – datos para esto pueden provenir de archivos descriptores definidos por el desarrollador, u obtenerse automáticamente de la base de datos.

Si los requerimientos de los DAOs son complejos, se puede considerar el uso de herramientas de terceros que provean el mapeo objeto – relacional para un DBMS. Estas herramientas generan el código una vez que el mapeo es completado y pueden incluir valores agregados como *cache* de resultados, *cache* de consultas, integración con servidores de aplicaciones, etc.

#### DAO *Factory*

Es posible hacer más flexible el patrón DAO adoptando los patrones *Abstract Factory* y *Factory Method*. Esta estrategia incluye un objeto DAO*Factory* que puede construir varios tipos de fábricas DAO, cada una soportando diferentes tipos de persistencia. Una vez que se obtiene el DAO*Factory* para una implementación específica, es usado para producir los DAOs necesarios de la aplicación.

## *Patrón Observer*

Proporciona a los componentes una forma flexible de enviar mensajes de difusión a los receptores interesados.

### *Estrategias*

Debido a que este patrón debe comunicar distintos componentes es necesario garantizar la unicidad en el envío de mensajes, evitando la redundancia o un paso de mensajes descontrolado. Esta característica implica dos situaciones que deben ser consideradas. La primera es que el componente *observer* sea único, lo cual se logra implementando el patrón *Singleton*, y la segunda se refiere al hecho que el *observer* deberá atender llamadas concurrentes por lo cual se tendrá que ejecutar un esquema multihilo y sin estado.

### *Regular* Java *Object*

El ciclo de vida de un objeto *observer* coincide con el de la aplicación, es decir, siempre que la aplicación se ejecute debe existir un objeto de este tipo monitoreando los cambios de estado de los objetos observados. Por esta razón es necesario que el objeto implemente los mecanismos multihilo.

### *Message – Driven Bean*

Un *message – driven bean* combina características de un *session bean* y de un oyente de mensajes de Java *Message Service* (JMS), permitiendo que un componente del negocio reciba mensajes JMS asincrónicamente. Esta estrategia resulta adecuada en un ambiente en el cual se requieren las características de un servidor de aplicación J2EE, pero demanda que se maneje la vida del componente de distinta manera al estar ligada a la sesión y no a la aplicación. Además obliga al uso de JMS para la comunicación.

## *Patrón Session Façade*

Puede ser usado un *bean* de sesión para encapsular las interacciones entre los objetos del negocio participando en un flujo de trabajo. La *session façade* administra los objetos del negocio, y provee de una capa uniforme de servicios de grano grueso a los clientes.

### *Estrategias*

Cuando se implementa la *session façade*, se debe decidir si será un *bean* de sesión con estado (*stateful*) o sin estado (*stateless*). Esta decisión debe basarse en el caso de uso que la *session façade* está modelando.

Durante la fase del análisis se debe poner atención sobre la naturaleza de las interacciones representadas en los casos de uso y sus escenarios. Identificar los estados mantenidos entre las interacciones (también conocidos como conversaciones) ayuda ha determinar el tipo de *bean* de sesión que se necesitará.

### *Stateless Session Façade*

Si el caso de uso es no-conversacional, entonces un solo método en la *session façade* lo inicia; cuando el método termina también lo hace el caso de uso. No hay necesidad de guardar el estado conversacional entre la invocación de un método y el siguiente. En este escenario la *session façade* puede ser implementada como un *bean* de sesión sin estado.

### *Stateful Session Façade*

Si el caso de uso es conversacional, entonces requiere múltiples llamadas a métodos para ser completado. El estado conversacional debe ser guardado entre cada invocación del cliente a un método; en este escenario un *bean* de sesión con estado es el enfoque más apropiado para implementar la *session façade*.

## *Business Objects*

Un *business object* se encarga de aplicar las reglas del negocio a los datos que participan en un flujo de trabajo. Este esquema mejora la separación de responsabilidades entre componentes de soporte y servicios, y los componentes que encapsulan al negocio. Además permite cambiar ciertas reglas sin alterar el resto de la aplicación.

### *Estrategias*

Es posible representar un objeto del negocio como un *bean* de sesión, un *bean* de entidad, un objeto de acceso a datos, o un objeto regular de Java. Las siguientes estrategias discuten cada una de estas opciones.

### *Session Bean*

Un *bean* de sesión normalmente provee un servicio del negocio y, en algunos casos, puede también contener datos. Cuando el *bean* necesita acceder a datos, usa un DAO para manipularlos. La *session façade* puede envolver uno, o más, de tales *beans*, actuando como objetos del negocio.

### *Entity Bean*

Representar los objetos del negocio como *beans* de entidad es el uso más común de la *session façade*. Cuando múltiples *beans* de entidad participan en un caso de uso no es necesario exponerlos al cliente, en su lugar la *session façade* los envuelve, y puede proveer un método para llevar a cabo la función requerida, ocultando la complejidad de las interacciones de los *beans*.

### *Regular* Java *Object*

La aplicación puede necesitar los servicios provistos por un objeto Java arbitrario, es decir, un objeto que no es un *session bean* ni un *entity bean* ni un DAO. En tales casos, la *session façade* accede a este objeto de servicio para proveer la funcionalidad necesaria.

## *Patrón Value Object*

El patrón *Value Object* es usado para encapsular datos del negocio. Sólo es necesario invocar un método para enviar u obtener un *value object*. Cuando el cliente solicita a un *enterprise bean* (*session* o *entity bean*) datos del negocio, el *enterprise bean* construye el *value object*, llena sus atributos, y lo pasa por valor al cliente.

### *Estrategias*

Las siguientes estrategias son aplicables cuando el *enterprise bean* es implementado como un *session bean* o como un *entity bean*. Estas estrategias son llamadas *Updateable Value Object* y *Multiple Value Object*.

En la primer estrategia el *value object* no sólo almacena los valores del objeto del negocio, sino que también transporta los cambios requeridos por el cliente hacia la entidad.

En algunos casos es posible que un solo objeto del negocio produzca diferentes *value objects* dependiendo de la petición del cliente. Esto sucede cuando existe una relación de uno a muchos entre el objeto del negocio y los *value object* que produce. En estas circunstancias la estrategia de *multiple value object* debe ser considerada.

### *Updateable Value Objects*

Esta estrategia es también conocida como *Mutable Value Object*. Recordemos que el cliente necesita leer y modificar los valores del *business object*, y para lograrlo se deben proveer métodos mutadores (estos métodos permiten modificar los valores de los atributos), también conocidos como métodos "*setXxx*". De esta manera debe existir un método "*set*" por cada atributo que requiera ser modificado. Sin embargo este tipo de métodos tienen las mismas limitaciones de desempeño, en términos de carga de la red, que los métodos "*get*".

En lugar de proveer métodos "*set*" individuales, el *business object* puede definir un solo método denotado por setData(*ValueObject*) que acepte al *value object* como parámetro, manteniendo los datos nuevos o modificados.

En el escenario del *value object* el cliente recibe una copia local a través del *business object*. Si el cliente quiere cambiar los valores sólo tiene que invocar los métodos "*set*" correspondientes. Este es un cambio local y no afecta los valores mantenidos por el *business object*. Cuando el cliente completa los cambios, solicita una llamada remota al *business object* a través del método setData(*ValueObject*), que serializa[12] la copia del cliente. El *business object* recibe

[12] Enviar como un flujo a través de un bus, como por ejemplo una red.

el objeto modificado e integra los cambios con sus propios atributos. Esta operación de integración puede complicar el diseño del *business object* y del *value object*, en términos de la propagación de los cambios, sincronización y control de versiones.

### *Multiple Value Objects*

Algunos *business object* pueden ser muy complejos, especialmente cuando son implementados como un *session bean*, aplicando el patrón *façade*. El *bean* interactúa con numerosos componentes del negocio para proveer un servicio. Así mismo el *session bean* puede producir su *value object* a partir de diferentes fuentes. En ambos casos se recomienda facilitar los mecanismos para producir *value objects* que representen partes de los componentes subyacentes de grano grueso. Un ejemplo de esto es un *session bean* que maneja datos de un cliente interactuando con otros *business objects* y componentes para proveer sus servicios. El *session bean* puede producir *value object* pequeños como la dirección del cliente o listas de contactos, que representan partes de su modelo.

### *Entity Inherits Value Object*

Para evitar la duplicidad de código el *entity bean* puede extender directamente (o heredar) la clase del *value object*, la cual implementa un método *getData*(), que al ser invocado en una instancia, crea una copia con todos los valores del objeto. Cuando el *entity bean* hereda del *value object*, un cliente puede invocar el método *getData*() y obtener los datos encapsulados.

De esta manera se elimina la duplicación de código entre el *entity bean* y el *value object*, y además ayuda a manejar los cambios en éste último previniendo que afecten al *entity bean*.

Esta estrategia tiene su costo relacionado con la herencia. Si el *value object* es compartido a través de este mecanismo, los cambios en su clase afectarán a todas sus subclases, obligando posiblemente otros cambios en la jerarquía.

### *Value Object Factory*

La estrategia anterior puede ser extendida para soportar múltiples tipos de *value objects* para un *entity bean*. Para lograrlo se debe definir una interfaz por cada tipo de *value object* que se deba regresar. La implementación *entity bean* de la superclase *value object* debe implementar todas estas interfaces. Además se deben crear clases separadas por cada interfaz definida.

Una vez que todas las interfaces han sido definidas e implementadas, se crea un método en el *value object factory* que recibe dos argumentos:

- La instancia del *entity bean* para la cual el *value object* debe ser creado.
- La interfaz que identifica el tipo del *value object* a crear.

El escenario que plantea esta estrategia inicia cuando el cliente pide el *value object* de un *business entity*. El *value object factory* recibe una llamada del *business entity* para crear un nuevo *value object*. El *business entity* pasa la clase del *value object* como un argumento, y el *value object factory* usa reflexión para obtener dinámicamente la información de la clase para el *value object* y construye una nueva instancia de éste. La obtención y actualización de valores del *business entity* por la *value object factory* es lograda utilizando invocación dinámica.

El costo asociado con esta estrategia está relacionado el uso de la reflexión contra el poder y la flexibilidad que otorga.

## *Framework Struts*

Conforme se desarrollan aplicaciones se identifica funcionalidad que es común. Normalmente se opta por repetir la solución para cada caso, pero esto además de consumir tiempo, aumenta la posibilidad de introducir errores. Otra opción, es crear una base común que brinde los elementos necesarios a todas las aplicaciones. Esta base constituye una aplicación semi – completa que debe ser especializada para lograr una solución particular. Actualmente existe un gran número de aplicaciones de este tipo y son conocidas como *frameworks*. Uno de ellos es *Struts* que surgió para apoyar aplicaciones web, haciendo uso de componentes J2EE.

Además de usar tecnologías basadas en Java, *Struts* hace uso de la arquitectura MVC (ver figura 4.1), proveyendo los componentes necesarios para las capas de vista y control. Por estas razones fue elegido como una de las tecnologías para implementar la arquitectura de HIM.

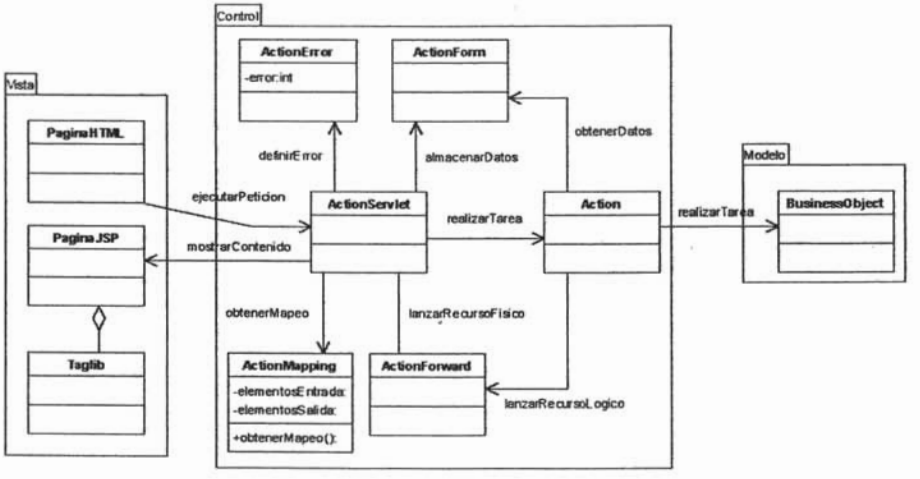



**Figura 4. 1**

A fin de comprender y aprovechar mejor la funcionalidad incluida en el *framework*, se efectuó la especificación de sus elementos, como son los *actions*, los *actionsForms*, el *actionServlet* y los archivos de configuración. Algunos de estos elementos son extendidos en la herramienta para completar la funcionalidad requerida (como el caso de los *action*), y otros sólo son invocados, o ligados a la aplicación, sin modificación (como el *actionServlet*).

Finalmente se incluyen los patrones de diseño implementados por *Struts*, describiendo el papel que juegan en el *framework*. Esto permite un mejor entendimiento de la capacidad de *Struts*, y constituyen un ejemplo de éxito en el uso de patrones, que puede ser seguido en otros desarrollos.

## *Arquitectura de Struts*

*Struts* implementa la arquitectura del Modelo 2 de Sun, el cual es una especificación con tecnologías J2EE (*JSPs* y *Servlets*) del patrón MVC. El Modelo 2 maneja las capas de vista y control, mientras que la capa del modelo queda abierta, para ser implementada a gusto del desarrollador con tecnologías como EJB, Spring, Hibernate, etc.

*Struts* incluye un *Servlet controller*, que es usado para manejar el flujo de navegación, y clases especiales para realizar el acceso a datos, de manera que las decisiones de control y flujo se mantengan fuera de la vista. Las páginas de la vista son manejadas con *JSPs*, a las que *Struts* añade bibliotecas de etiquetas personalizadas para facilitar las tareas comunes.

Como se mencionó anteriormente *Struts* está basado en *JSPs* y *Servlets*, y por lo tanto requiere de un entorno de ejecución. Esto se debe a que los ciclos de vida de los componentes deben ser administrados por un servidor de aplicación (entorno).

### *Componentes*

Entre los principales componentes de *Struts* se tienen:

- *actionForward*
- *actionForm*
- *actionMapping*
- *actionServlet*
- *action*
- Bibliotecas de etiquetas personalizadas

El *actionServlet* (implementación del *servlet controller*) controla la navegación. Otra clase de *Struts*, el *action*, es usada para acceder a las clases del negocio. Cuando el *actionServlet* recibe una petición a través del servidor de aplicación, emplea la URI para determinar qué *action* se usará. El *action* puede validar la entrada y acceder la capa del negocio para obtener información de una base de datos, o de otros servicios.

Para validar la entrada, el *action* necesita conocer los valores enviados. En vez de que cada *action* los obtenga directamente de la petición, el *actionServlet* encapsula la entrada en un *JavaBean*. Estos *beans* de entrada son subclases de la clase *action*Form. El *actionServlet* puede determinar qué *action*Form usar analizando la petición, de la misma manera en que el *action* fue seleccionado.

Normalmente, el *action* no genera la respuesta sino que redirecciona la petición a otro elemento, como una página *JSP*. *Struts* provee la clase *actionForward* que puede ser utilizada para almacenar la ruta de la página bajo un nombre lógico. Cuando se completa el proceso del negocio, el *action* selecciona y regresa un *actionForward* al *actionServlet*, el cual usa la ruta almacenada para llamar la página indicada y completar la respuesta.

*Struts* encapsula estos detalles en un objeto *actionMapping*, el cual está asociado a una ruta específica. Cuando esa ruta es invocada, el *servlet* obtiene el objeto *actionMapping*, que le indica qué *actions*, *actionForms* y *actionForwards* usar.

Todos estos detalles, los *actions*, los *actionsForms*, *actionForwards*, *actionMappings* son declarados en el archivo *Struts-config*.xml. El *actionServlet* lee este archivo al iniciar la aplicación y crea una base de datos con los objetos de la configuración.

La siguiente tabla muestra las principales clases de *Struts* y resume sus responsabilidades dentro del MVC.

| Clase | Descripción |
|---|---|
| *actionForward* | Maneja las acciones del usuario y la selección de vistas. |
| *actionForm* | Contiene los datos necesarios para un cambio de estado. |
| *actionMapping* | Maneja el evento de un cambio de estado y almacena las rutas lógicas. |
| *actionServlet* | Es la parte del *controller* que recibe las acciones del usuario y los cambios de estado y lleva a cabo la selección de vistas. |
| *action* | La parte del *controller* que interactúa con el modelo para ejecutar un cambio de estado o consulta, y notifica al *actionServlet* la siguiente vista a seleccionar. |

Para presentar los datos en la vista, el *framework* provee de varios *helpers*[13] en la forma de etiquetas *JSP*:

| Descriptor de la biblioteca | Propósito |
|---|---|
| *Struts*-html.tld | Extensión de etiquetas *JSP* para formas HTML. |
| *Struts-bean*.tld | Extensión de etiquetas *JSP* para el manejo de *JavaBeans*. |
| *Struts-logic*.tld | Extensión de etiqueta *JSP* para evaluar los valores de las propiedades. |

Estos elementos en su conjunto forman los módulos del patrón MVC:

| Vista | Control | Modelo |
|---|---|---|
| Extensiones de etiquetas *JSP* | *actionForward*<br>*action*Form<br>*actionMapping*<br>*actionServlet*<br>Clases *action*<br>*actionErrors* | Fuentes de datos genéricas. |

***El flujo de control***

La figura 4.2 muestra el proceso de petición – respuesta de *Struts*. Los números en los paréntesis corresponden con los números en las invocaciones.

1. Un cliente solicita una dirección que está mapeada a la URI de un *actionServlet* (1).
2. El contenedor (servidor de aplicación) pasa la petición al *actionServlet*.
3. El *actionServlet* busca el mapeo correspondiente a la dirección solicitada.
4. Si el mapeo especifica un *bean* de forma (datos de pantalla), el *actionServlet* revisa si ya existe alguno, en caso contrario, lo crea (1.1).
5. El *actionServlet* limpia los campos del *bean* y asigna los valores obtenidos de la petición HTTP.
6. Si el mapeo tiene la propiedad *validate* marcada como *true*, se invoca la validación en el *bean* (1.2).
7. Si la validación falla el *Servlet* cambia el control a la ruta especificada en la propiedad *input*, y el flujo termina.
8. Si el mapeo especifica un tipo de *action*, es reutilizado en caso de existir o se crea uno nuevo (1.3).

[13] Ver el patrón *View Helper* en el capítulo Patrones para la Arquitectura de HIM.

9. Se invoca el método *perform* o *execute* del *action* y se le pasa la instancia del *bean*.
10. El *action* puede cambiar los datos del *bean*, invocar objetos del negocio, o hacer cualquier cosa que sea necesaria (1.3.1 – 1.3.4).
11. El *action* regresa un *actionForward* al *actionServlet* (1.3.5).
12. Si el *actionForward* especifica otro *action* el proceso empieza de nuevo, de otro modo se puede invocar una página o algún otro recurso, frecuentemente un *JSPs* (2 y 3).

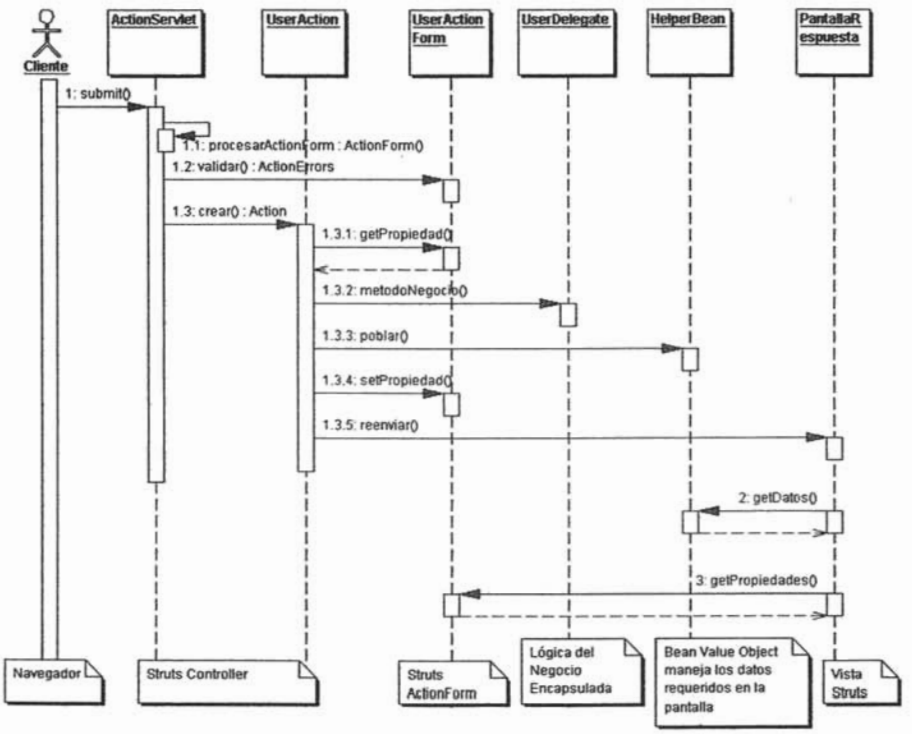



**Figura 4.2**

Autor: Jean Pierre Garnier
http://rollerjm.free.fr

## *Patrones de Struts*

### El patrón *Service* to Worker

En el nivel más alto (ver figura 4.3), *Struts* implementa el patrón *Service* to Worker. Este es un "macro" patrón que incorpora a otros dos: *Front Controller* y *View Helper*.

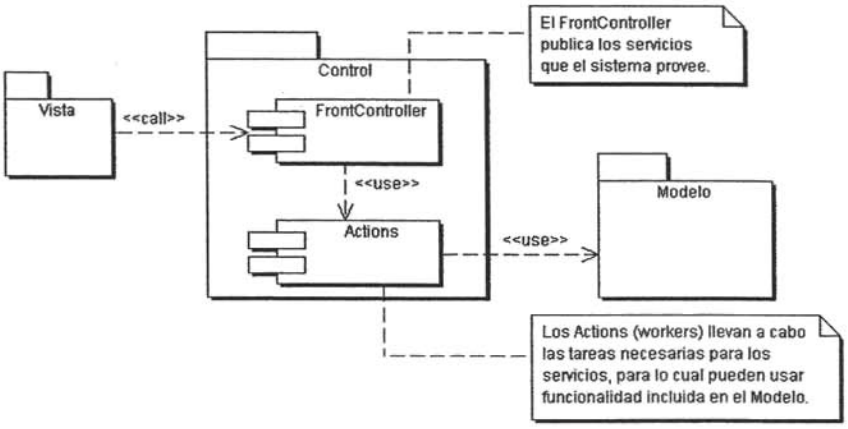



**Figura 4. 3**

### El patrón *Front Controller*

Como se describió anteriormente, el *actionServlet* recibe las acciones del usuario y establece los cambios. El *actionServlet* proporciona de un punto de acceso centralizado para el manejo de peticiones, incluidos los servicios del sistema, de seguridad, la obtención de contenido, la administración de la vista, y la navegación, de manera tal que el código no es duplicado, ni mezclado con el contenido de la vista.

### Los patrones *Command* / *Command* and *Controller*

*Struts*, como muchos *front controllers*, implementa un componente *dispatcher* (la clase *action*) para manejar detalles como la obtención de contenido y la administración de la vista. El *actionServlet* invoca un método en el *action* y pasa los detalles de la petición para delegar responsabilidades de la respuesta. Esto es conocido como la estrategia *Command* and *Controller* [Alur], y esta basada en el patrón *Command* [Gamma].

## El patrón *Service* Locator

Gran número de componentes de persistencia se basan en la clase DataSource (incluida en el paquete javax.sql de las bibliotecas de Java), para el acceso al sistema de almacenamiento. Frecuentemente el almacenamiento es una base de datos JDBC, pero un DataSource puede ser usado para conectarse a muchos tipos de sistemas. El *actionServlet* manteniene una lista de objetos DataSource referenciados por nombres lógicos, a partir de la configuración de *Struts*. De esta manera otros objetos pueden obtener el DataSource (localizar el servicio) por el nombre sin tener que conocer ningún detalle de implementación. Los desarrolladores de *Struts* pueden cambiar controladores JDBC, pools de conexiones, u otros componentes con sólo modificar el DataSource registrado.

## El patrón *View Helper*

El *framework* espera que los datos que son ingresados en el sistema sean encapsulados en *JavaBeans*. Una de las cosas que *Struts* administra es la recepción de peticiones HTTP y la transferencia de sus parámetros a un *bean actionForm*. El uso de *JavaBeans* para transferir datos del negocio a componentes de la vista, se conoce como el patrón *View Helper*. El *framework* de *Struts* pone a disposición del desarrollador varios recursos en el contexto del *Servlet*, entre ellos los *JavaBeans*, de manera que pueden ser usados por otros componentes, especialmente por aquellos de la capa de presentación.

## El patrón *Singleton*

En un ambiente multitarea como Java, pueden haber desventajas al tener múltiples copias de un mismo objeto. Los *Servlet*s, por ejemplo, son multihilo y pueden manejar cualquier número de peticiones sin lanzar nuevos objetos. *Struts* aplica este mismo concepto a sus *action*s y crea únicamente un *action* por aplicación. Esto mejora el desempeño, pero responsabiliza al desarrollador de escribir *action*s de hilo seguro.

## El patrón *Session Façade*

La interacción entre el modelo y el resto de la aplicación puede llegar a ser muy compleja. Encapsular estos detalles en un objeto con una interfaz única puede hacer que una aplicación sea más fácil de escribir y mantener. *Struts* promueve encapsular tales detalles en sus clases *action*, las cuales contienen una interfaz bien definida con algunas responsabilidades generales, pero pueden incluir cualquier funcionalidad necesitada por el sistema.

Este esquema es conocido como el patrón *Session Façade*: la clase *action* "abstrae la interacción de los objetos del negocio subyacentes y provee una capa de servicios que expone únicamente las interfaces requeridas" [Alur].

*Struts* también usa el patrón de *Session Façade* en el diseño del componente para los mensajes de las pantallas. Se oculta la complejidad de seleccionar la localización del usuario (el lugar donde se encuentra, por ejemplo su país). El componente simplemente solicita el mensaje por su clave, y el proveedor de mensajes devuelve el mensaje del contexto correcto.

## Los patrones *Value Object* y *Value Object* Assembler

Actualmente, con las interfaces de usuario basadas en cuadros de diálogo, es posible solicitar o enviar un grupo de datos de una sola vez. Esto toma especial importancia cuando la aplicación usa un servidor remoto, y existe algún retardo entre las peticiones, haciendo necesario obtener toda la información posible cada vez.

En el ambiente de J2EE, la información que se necesita agrupar en un único paquete es llamada *value object*, el cual encapsula datos del negocio, de manera que puedan ser enviados o recibidos a través de una única llamada [Alur]. El *action*Form de *Struts* es un ejemplo de *value object*, que se encarga de recolectar los campos necesarios desde una forma HTML, de manera que la información puede ser validada en un solo paso, y regresada para ser corregida, o enviada para ser procesada por un *action*.

Los *actionErrors* son otro ejemplo de *value object* que permiten recolectar varios mensajes de error de modo que puedan ser mostrados al mismo tiempo, corregidos simultáneamente, y reenviados.

Los *action*Forms pueden usar *JavaBeans* anidados lo que permite dividir las propiedades requeridas en diferentes objetos, obtenidos de diferentes partes del modelo. Esto es conocido como el patrón *Value Object* Assembler, en el cual el conjunto de datos puede ser tratado como una entidad única.

## El patrón *Composite View*

Frecuentemente una pantalla de presentación es hecha a partir de varias pantallas individuales: *Composite View*. Las etiquetas de *Struts* 1.0 Template usan este patrón para construir un *JSP* desde distintas páginas. En general, esto es usado para proveer a la página de un conjunto de elementos comunes de navegación, los cuales pueden ser cambiados con la edición de una sola página.

## El patrón Synchronizer

Un problema frecuente en las aplicaciones web es el retardo entre las peticiones y sus respuestas. Esto puede ser un serio inconveniente cuando una petición sólo debe ser enviada una sola vez. Este es el caso, por ejemplo, cuando un usuario envía una orden de compra y debido a que este trámite puede tomar algunos minutos antes de recibir una respuesta, es fácil que el usuario se

impaciente y presione nuevamente el botón de envío, generando nuevamente una solicitud, y duplicando la orden.

Una estrategia para prevenir esta situación es colocando un campo sincronizador en la sesión del usuario, e incluir el valor del campo en la forma que se necesita proteger. Cuando la forma es enviada el *action* limpia el campo, y si envíos posteriores encuentran que falta, el *action* podrá ignorar el envío duplicado mostrando además un mensaje al usuario donde se le indique que su orden está siendo procesada.

La clase *action* de *Struts* implementa la estrategia de campo sincronizador, el cual también se presenta en las etiquetas de extensión.

## El patrón *Decorator*

Este patrón proporciona una forma flexible de añadir funcionalidad a un componente sin modificar su apariencia externa o su función.

El objeto *action* es un *singleton* multihilo. Esto, en general es deseable, pero hace difícil reutilizar el *action* para tareas similares. La clase *actionMapping* es usada para extender objetos *action*, en lugar de crear una clase que herede de ella. Un *actionMapping* le da responsabilidades adicionales, y nueva funcionalidad, lo cual incorpora al patrón *Decorator*.

# Capítulo 5

# Arquitectura de HIM

Con base en los patrones que se han descrito se propone una arquitectura de diseño para HIM. La especificación de la arquitectura incluye una descripción general que brinda una visión amplia de la herramienta, su capacidad y su estructura.

## *Componentes*

Basados en los paquetes especificados en la etapa del análisis[14], se estableció qué entidades (ver figura 5.1) serían manejadas por los módulos del patrón MVC. De este modo la capa de interfaz humana, será manejada a través de páginas web dinámicas, en el módulo de la vista. La capa del dominio del problema, constituida por los paquetes de control y seguridad, se incluyeron como parte del módulo de control. Finalmente la capa de manejo de datos, que incluye el paquete de base de conocimiento formó el módulo del modelo.

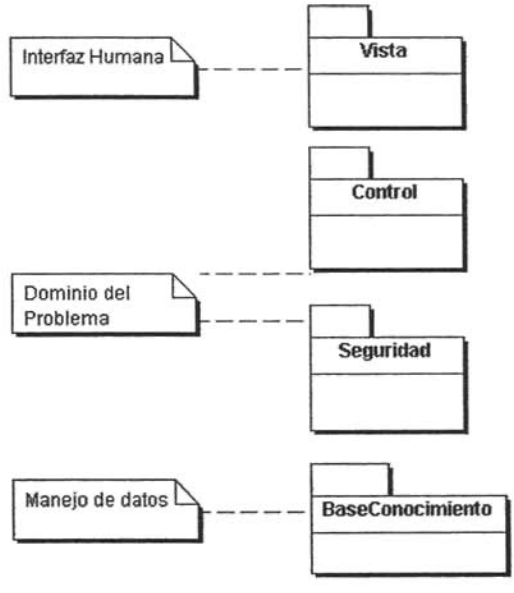



**Figura 5. 1**

Una vez definida la responsabilidad de los módulos se procedió a diseñar el funcionamiento interno de cada uno. A continuación se comentan los patrones usados y la función que cumplen en la arquitectura.

[14] Ver documento Diagramas de Clases del Análisis, incluido en el CD de HIM.

En la vista, ver figura 5.2, se aprovechó el manejo dinámico de las pantallas proporcionado por el patrón *View Helper* que delega la obtención del contenido a objetos *helpers*, que sirven como modelo de los datos y se encargan de adaptarlos para ser presentados. Otro patrón utilizado es el *Composite View*, que define plantillas genéricas para las pantallas, que son personalizadas en tiempo de ejecución dependiendo del estado del sistema que se requiera presentar al usuario.

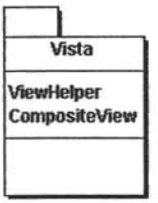



**Figura 5. 2**

Para la capa de control, ver figura 5.3, se utilizará el patrón *Front Controller*[15], el cual simplifica la comunicación entre los objetos de las vista y del control, introduciendo un único objeto que gestiona la distribución de mensajes. Una variación de este patrón contempla el manejo de roles configurables, en el cual los clientes definen un rol (pudiendo ser cambiado conforme se ejecuta el sistema) que tendría requerimientos para los mensajes. Otro patrón empleado es el *Filter*[16], el cual aplica una serie de validaciones (filtros) a las peticiones o eventos lanzados por los usuarios a través de la interfaz. Este mecanismo es dinámico y permite la asignación de validaciones a peticiones específicas sin tener que modificar el código existente, gracias a archivos de configuración.

El segundo paso en el proceso de una petición es asignar los objetos que van a realizar la tarea solicitada, lo cual se obtiene mediante el patrón *Command*. Este patrón permite crear en tiempo de ejecución, los objetos apropiados para responder a una tarea. Entre las ventajas provistas por este patrón está el encapsular un comando en un objeto de forma que puede ser almacenado, pasado como parámetro o devuelto por un método. Esta característica provee al sistema de la ayuda necesaria para deshacer comandos, el manejo de la seguridad al nivel de petición, procesos de identificación, manejo de transacciones, establecer colas de comandos, y desacoplar la fuente de una petición del objeto que la cumple.

[15] Ver las secciones *Tecnologías y Estrategias para el Control* y *Patrones de Struts* del capitulo Tecnologías y Estrategias para Patrones.
[16] Este patrón no es descrito en esta tesis debido a su alcance pero para mayor referencia ver [Vázquez].

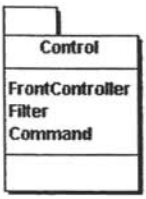



**Figura 5. 3**

Por su parte, la capa del modelo, ver figura 5.4, se encarga de la persistencia de los datos, la consistencia de éstos y la creación de los objetos del negocio. La persistencia es implementada con RDF y XML, a través de bibliotecas de Jena, y directamente en el sistema de archivos del servidor. Para la traducción de los datos desde RDF a objetos se usó el patrón DAO (*Data Access Object*), que permite desacoplar la fuente de los datos de los objetos del negocio. Esto facilita la inserción de distintos esquemas de almacenamiento, como puede ser una base de datos relacional.

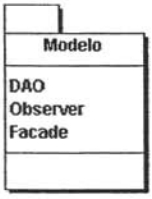



**Figura 5. 4**

Debido a que se requiere manejar información interrelacionada a distintos niveles: documentos, modelo del negocio, meta información y cada uno de estos niveles está implementado de distinta manera, es necesario contar con un mecanismo que garantice la consistencia de los datos. Para lograrlo se utilizó el patrón *Observer*, que permite tener un medio centralizado que capte las modificaciones en los módulos de datos y reporte los cambios a las partes afectadas, actualizándose apropiadamente. Este módulo se llamó Sincronizador [Mercado], y para garantizar la consistencia debe ser único, es decir, como objeto del sistema sólo puede instanciarse una vez para evitar la duplicación de mensajes. Este comportamiento de unicidad se logra con el patrón *Singleton*.

Finalmente se usó el patrón *Façade* para proveer una clase para la interacción entre el control y el modelo. La clase que realiza esta función recibe peticiones sobre los objetos del negocio y devuelve objetos con valores (*value objects*), que son un reflejo de los objetos del negocio pero únicamente contiene datos y métodos para leer o escribir en ellos.

HIM implementa el patrón de arquitectura MVC. La vista y control son manejadas con el *framework Struts*, mientras que el modelo es renombrado como Base de Conocimiento (BC), y se apoya en la tecnología RDF [Hernández]. Como

parte de la arquitectura se explican los elementos que la componen (ver la figura 5.5), los cuales pertenecen a tecnologías y patrones incluidos en el capítulo anterior.

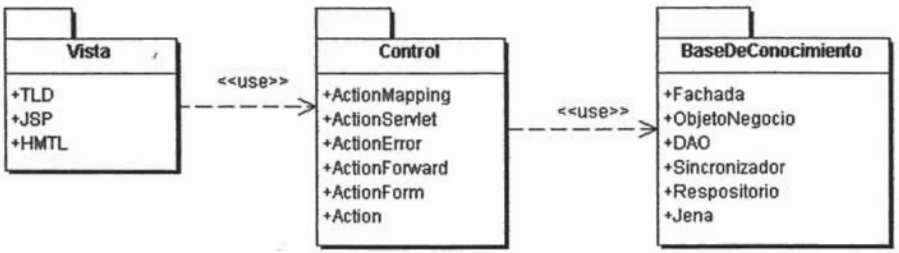



**Figura 5. 5**

## *Base de Conocimiento*

La BC constituye un módulo del sistema que sólo puede ser accedido a través de una clase que funciona como fachada, la cual expone, únicamente, la funcionalidad requerida por el resto de la aplicación. Internamente la BC incluye un gran número de componentes que permiten la persistencia, sincronización y estructuración de los datos [Mercado].

Para lograr la persistencia se utiliza RDF y un repositorio en el sistema de archivos. El módulo de RDF maneja el almacenamiento de la información cuyo contenido es necesario explotar: procesos, proyectos, actividades, meta información, productos PIEs, datos de usuarios, entre otros. Esta información está implementada en XML y para explotarla se hace uso de las bibliotecas de Jena, a través de la clase del mismo nombre.

El repositorio se usa para almacenar documentos como un todo sin opción de manipular los datos que contienen. Estos documentos son llamados CajasNegras y pueden estar en cualquier formato. La aplicación únicamente registra datos descriptivos sobre estos documentos, como su nombre, estado y versión.

Sobre la persistencia se implementa una capa de objetos de acceso a datos (DAO), la cual se encarga del mapeo de objetos a entidades RDF, y viceversa. Las clases que residen en esta capa usan Jena, de modo que no es necesario trabajar a bajo nivel con documentos XML, si no que se hace referencia a las tríadas que constituyen la información en RDF. Esta misma capa se encarga de encapsular los documentos que son almacenados en el repositorio, de manera que la aplicación los pueda usar como objetos Java.

Debido a que la herramienta requiere almacenar información de una misma entidad, por ejemplo un producto CajaNegra, en dos medios distintos (Repositorio y RDF), es necesario garantizar la consistencia de los datos. Para lograr esto se implementa un sincronizador. Este módulo se encarga de notificar a los componentes que mantiene registrados, los cambios que puedan ocurrir en alguno de ellos. La clase que implementa el sincronizador está construida para sólo permitir la existencia de un objeto a la vez, de modo que no pueda haber distintas instancias intentado sincronizar los mismos componentes.

Los datos obtenidos de la persistencia son encapsulados en *business objects*. Prácticamente existe un *business object* por cada entidad de MoProSoft. Estos incluyen los procesos, proyectos, roles, actividades y productos. Adicionalmente se han incluido las clases MarcoTrabajo, MarcoGeneral y Esquema, las cuales son necesarias para manejar los *business objects* en las capas de control y vista.

***Fachada***

La clase BaseConocimiento incluye la funcionalidad que la BC expone a la capa de control. Esta clase implementa el patrón *Façade* ya que sirve como interfaz entre distintas capas y representa un único punto de interacción.

La figura 5.6 presenta el diagrama con las clases de fachada y las de intercambio de datos:

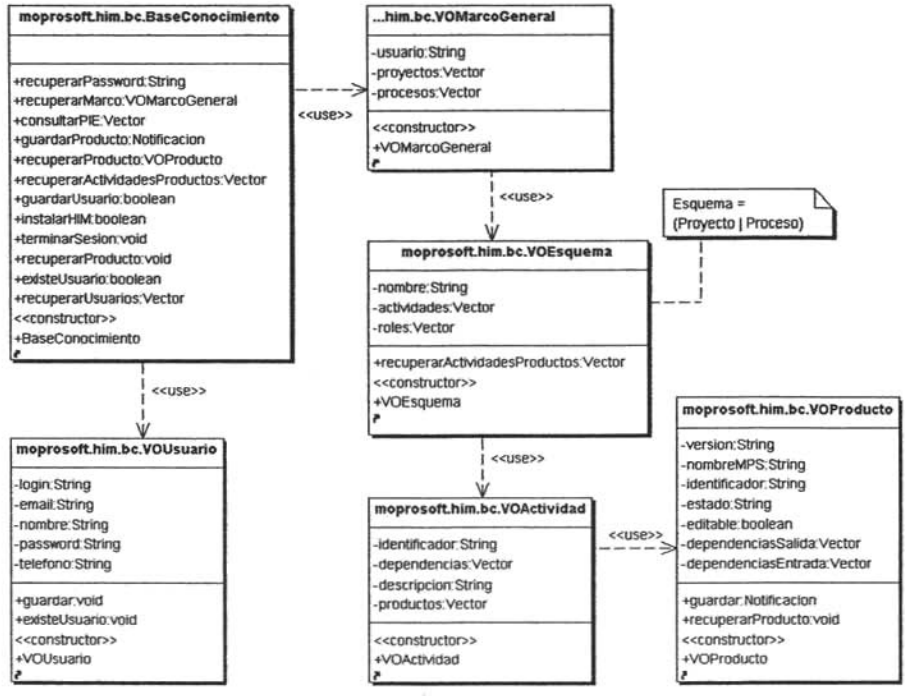



**Figura 5.6**

Para cumplir con una tarea, la clase BaseConocimiento invoca los *business objects* que incluyen la funcionalidad requerida, y una vez que estos terminan su proceso, devuelven los datos obtenidos encapsulados en un *value object* (se identifican por el prefijo VO en el nombre de la clase). Por cada *business object* existe un VO, que únicamente contiene datos y métodos de lectura / escritura. Esto garantiza que los objetos que son pasados a la capa de control no tengan acceso a funcionalidad interna de la BC, y por lo tanto, no puedan violar la seguridad que proporciona la fachada. Adicionalmente este esquema de VOs permite disminuir el tamaño de los objetos que manejará la capa de control.

Los VOs que se tienen son:

- *VOUsuario*. Encargado de almacenar el *login* y *password*, así como datos personales del usuario.
- *VOMarcoGeneral*. Almacena los procesos y proyectos el los cuales el usuario tiene asignado un rol.
- *VOEsquema*. Representa a un proceso o proyecto, e incluye sus actividades y los roles del usuario.

- *VOActividad*. Encapsula los datos de una actividad incluidas las dependencias que pueda tener hacia otras actividades, y los productos que genera.
- *VOProducto*. Incluye datos necesarios para el usuario, como el nombre del producto y sus dependencias, y datos para su manejo, como su estado y un identificador.

### *Marco General*

La clase MarcoGeneral está encargada de reunir los procesos y proyectos con los que está relacionado un usuario. Los objetos de esta clase son invocados por el método recuperarMarco() de la clase BaseConocimiento.

La figura 5.7 muestra el diagrama de la clase MarcoGeneral, junto con las clases que requiere para llevar a cabo sus tareas. Como se puede observar se utilizan las clases DAORDFProceso, DAORDFProyecto y JENA, de la capa de acceso a datos.

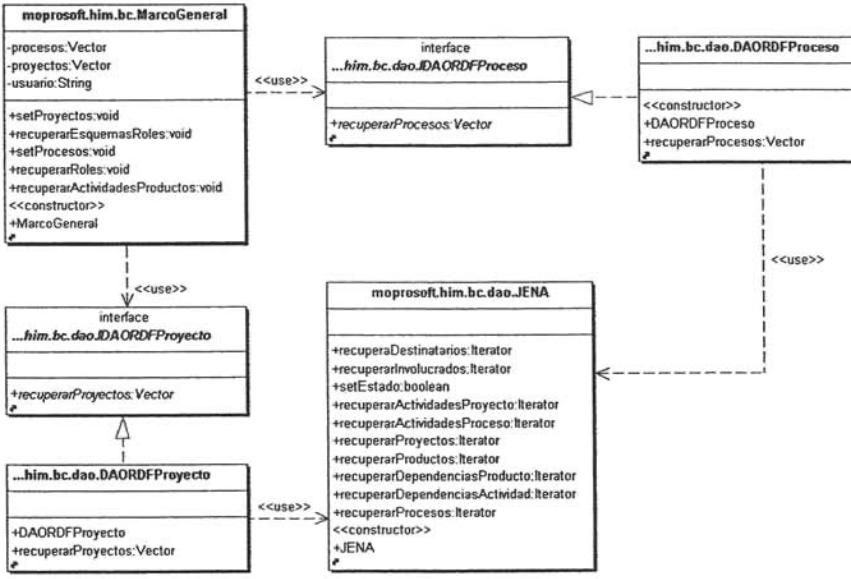



**Figura 5.7**

Las clases DAORDFProceso y DAORDFProyecto implementan las interfaces IDAORDFProceso y IDAORDFProyecto respectivamente, de manera que se obtiene un bajo acoplamiento entre ellas y el MarcoGeneral. Todas las clases con el prefijo "DAORDF" hacen uso de la clase JENA, la cual provee el acceso a los recursos RDF.

Para formar el marco general (ver figura 5.8) de un usuario es necesario conocer además de los procesos y proyectos, los roles (ver figura 5.9) que juega en ellos. Una vez que el usuario tiene a su disposición esta información y elige uno en particular se deben proveer las actividades y productos asignados a sus roles en ese proceso o proyecto.

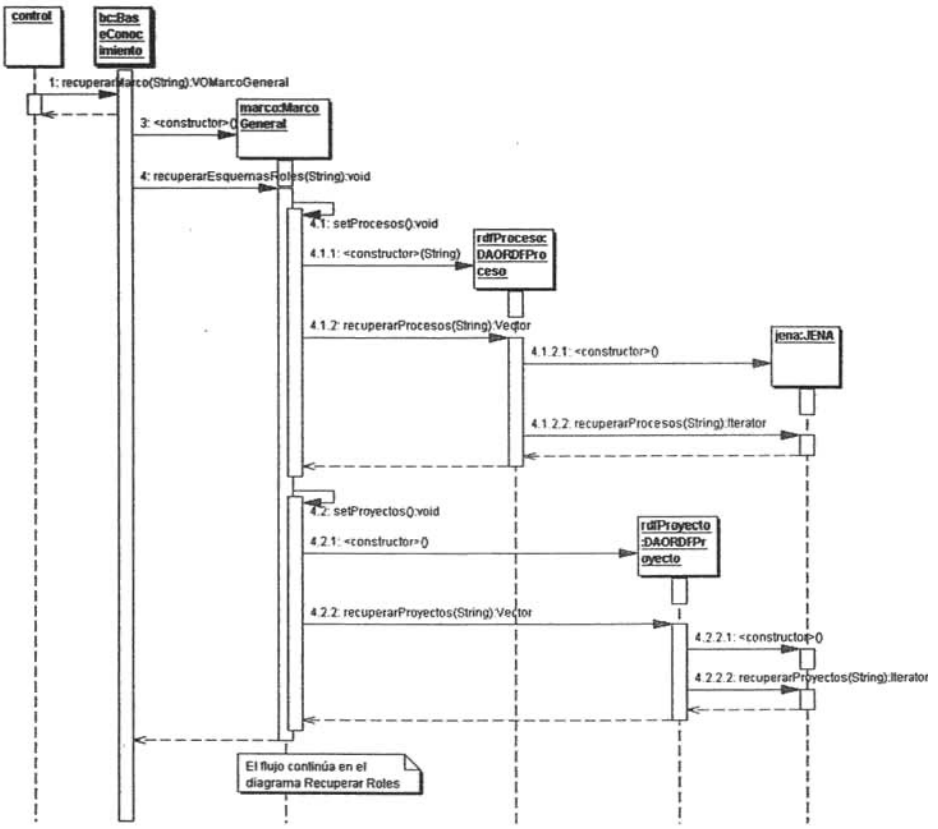



**Figura 5. 8**

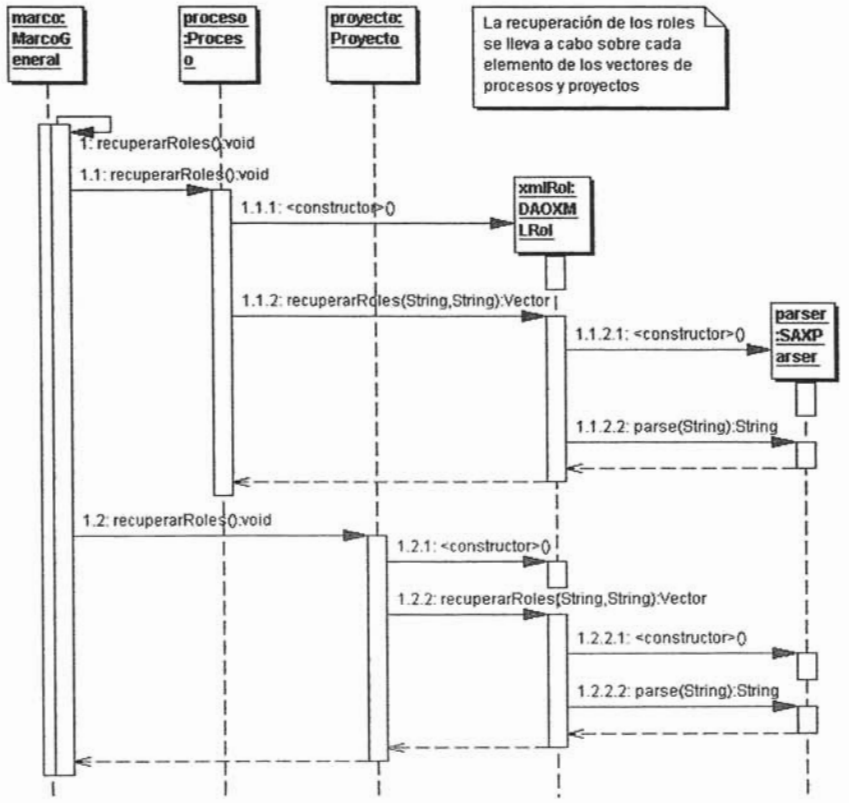



**Figura 5. 9**

### *Esquema*

La figura 5.10 muestra la clase Esquema definida como una generalización de las clases Proyecto y Proceso, de manera que los elementos comunes de estas dos clases son factorizados. La clase MarcoGeneral hace uso de Esquema.

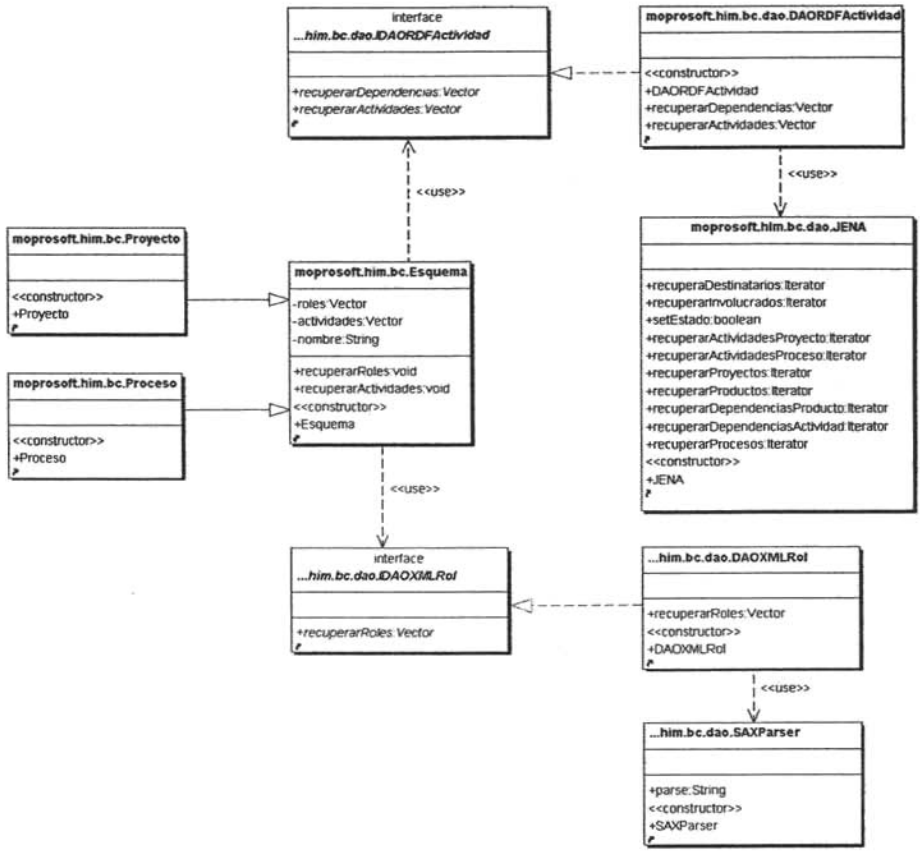



**Figura 5.10**

Las clases relacionadas con el esquema son:

- *DAORDFActividad*. Comentada en la descripción de la clase Actividad.
- *DAOXMLRol*. Encargado de recuperar los roles que tiene la actividad. En este caso cambia el prefijo a DAOXML para indicar que los datos están almacenados en XML en lugar de RDF.
- *SAXParser*. Esta clase realiza el acceso a documentos XML, y provee los datos requeridos por las clases con el prefijo a DAOXML.

### *Actividad*

La figura 5.11 muestra la clase Actividad que da soporte a las actividades de MoProSoft. Para lograr esto incluye una descripción de la actividad, los productos manejados, y las dependencias con otras actividades. Los datos de las

actividades, como son la descripción y las dependencias son almacenadas en RDF, para lo cual se hace uso de la clase DAORDFActividad.

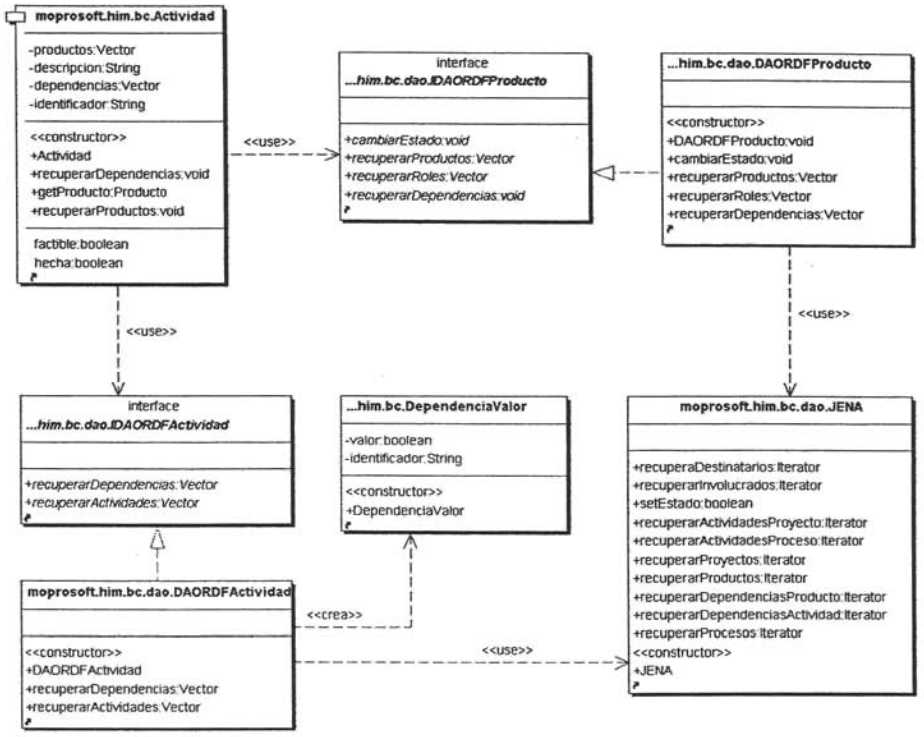



**Figura 5.11**

El atributo factible sirve para indicar si la actividad puede ser realizada por el usuario actual, lo cual depende de sí se han realizado las actividades de las cuales depende la actividad en cuestión o de sí el usuario que la invocó cuenta con el rol adecuado.

Los elementos que apoyan a la clase Actividad son:

- *DAORDFActividad*. Recupera los datos de la actividad desde la capa de persistencia.
- *DependenciaValor*. Encapsula las dependencias de la actividad e indica si han sido realizadas.
- *DAORDFProducto*. Obtiene los productos y los roles que el usuario puede manejar en la actividad.

En el siguiente diagrama (ver figura 5.12) se muestra la interacción de las clases mencionadas para obtener las actividades.

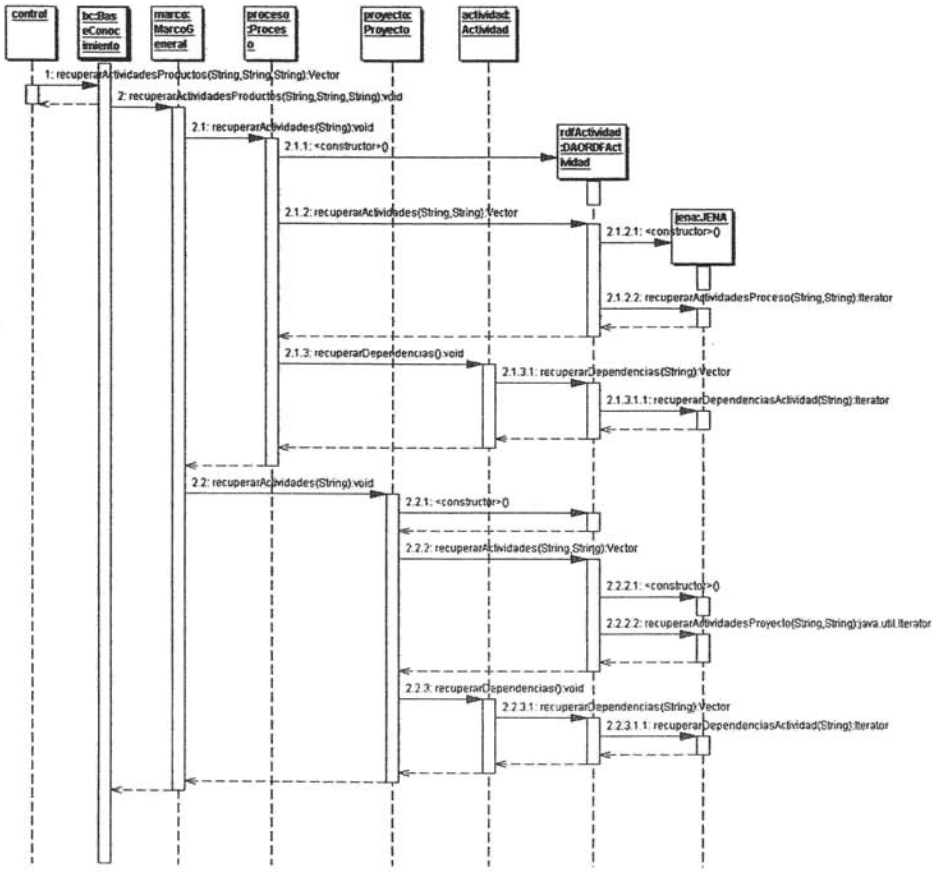



**Figura 5. 12**

### *Producto*

La clase Producto, mostrada en la figura 5.13 representa a los documentos que son obtenidos a partir de las actividades. El Producto contiene información incluida en MoProSoft, así como datos de control y soporte para manejar sus dependencias hacia otros productos, las cuales pueden ser de dos tipos:

- *De entrada*, cuando para trabajar con un producto es necesario haber creado otros previamente.
- *De salida*, cuando hay productos que dependen del actual para ser realizados.

El producto maneja distintos estados dependiendo de las fases por las que va pasando en su realización. Dependiendo del cambio de estado se debe notificar a los usuarios, que por sus roles están relacionados con el producto.

Para almacenar el producto es necesario registrar la ruta donde se localiza. Con cada modificación que lo afecta se genera una nueva versión de manera que puedan ser rastreados los cambios realizados.

A continuación se presentan los diagramas con las clases utilizadas por el producto para cumplir con sus responsabilidades. En el primer diagrama, figura 5.13, se muestra la clase Producto junto con las clases que heredan de ella, así como las clases de las cuales dependen directamente. En el segundo diagrama, figura 5.14, aparecen las clases que sirven de apoyo a las mostradas en el diagrama anterior, y que realizan el acceso a los recursos de la capa de persistencia.

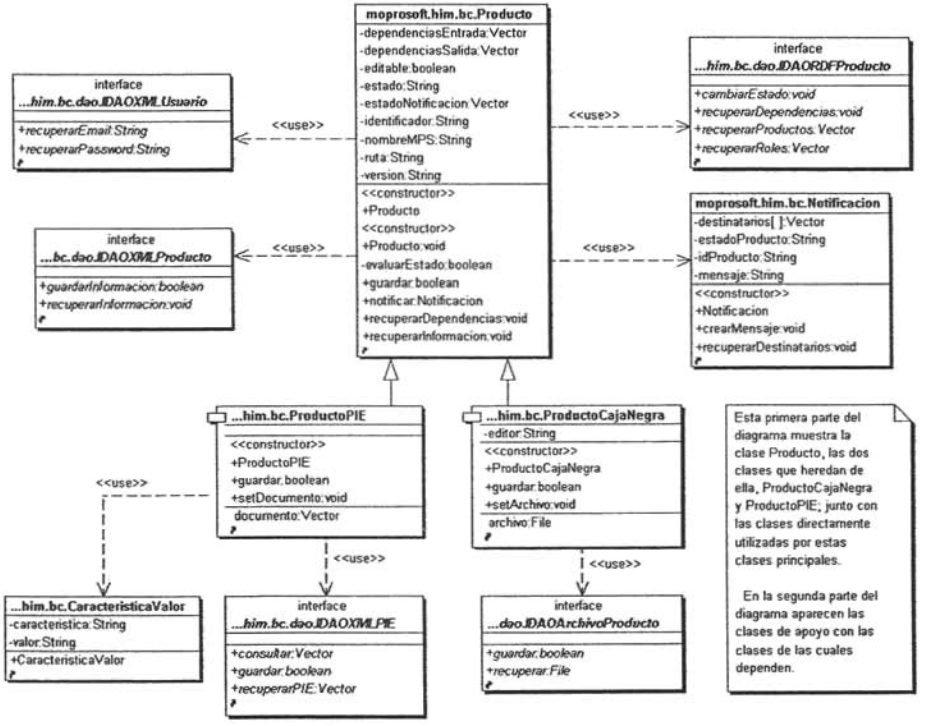



**Figura 5.13**

Los productos son especializados en las clases ProductoPIE y ProductoCajaNegra. Los primeros dan soporte a aquellos documentos cuyo contenido deberá almacenarse en detalle, para poder explotar sus datos. Por su

parte, los ProductoCajaNegra son responsables de los documentos que sólo son almacenados como un todo sin necesidad de conocer sus datos internos.

Las siguientes clases e interfaces son utilizadas por los productos:

- *IDAOXMLUsuario*. Define las responsabilidades que se deben implementar para que el producto notifique a los usuarios.
- *IDAOXMLProducto*. Define las responsabilidades en la recuperación y guardado de datos descriptivos del producto, en XML.
- *CaracteristicaValor*. Encapsula los datos de los PIEs en pares ordenados, donde la primera posición representa el nombre de algún campo de producto y la segunda contiene el valor de ese campo.
- *IDAOXMLPIE*. Define la funcionalidad para manejar la información de los PIEs. Esta interfaz fue modificada en la implementación para usar recursos RDF, para mayor información referirse a [Hernández].
- *IDAOArchivoProducto*. Define las responsabilidades para el manejo de los productos del tipo "caja negra".
- *IDAORDFProducto*. Define las responsabilidades en la recuperación y guardado de datos en RDF, como las dependencias y el estado.
- *Notificación*. Permite el envío de mensajes a los usuarios que por sus roles deben conocer el cambio de estado de un producto.

En este segundo diagrama se muestran las clases que implementan las interfaces incluidas en el diagrama anterior (figura 5.13). Además se tienen las clases que sirven de apoyo para la capa de persistencia: DAOArchivoProducto, SAXParser y JENA.

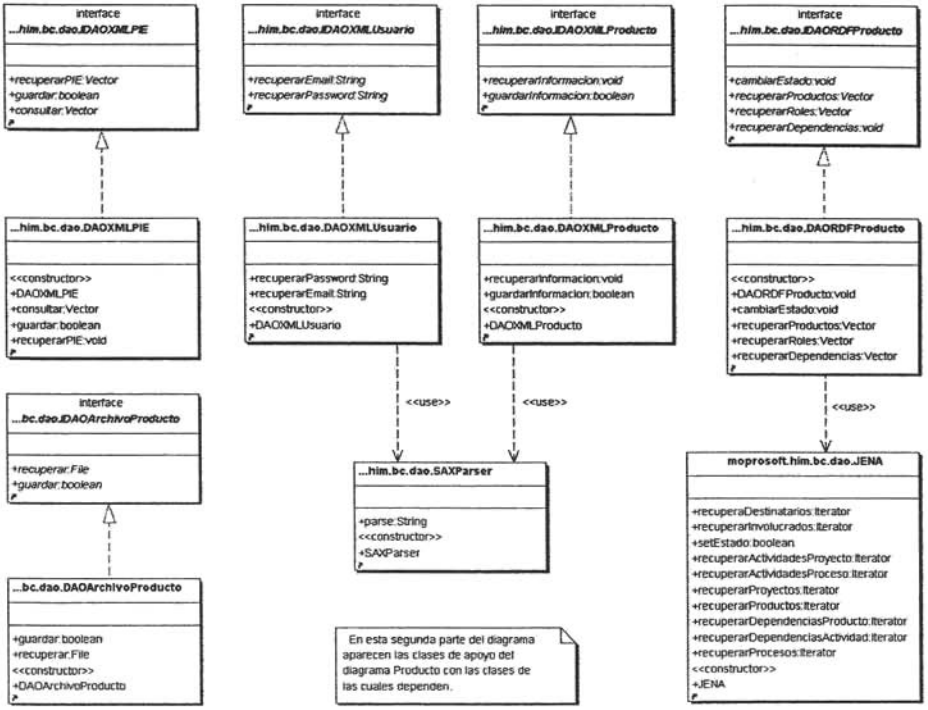



**Figura 5.14**

En el siguiente diagrama, figura 5.15, se muestra la obtención de los productos a partir de un proceso o un proyecto, y en el diagrama de la figura 5.16 a partir de una actividad.

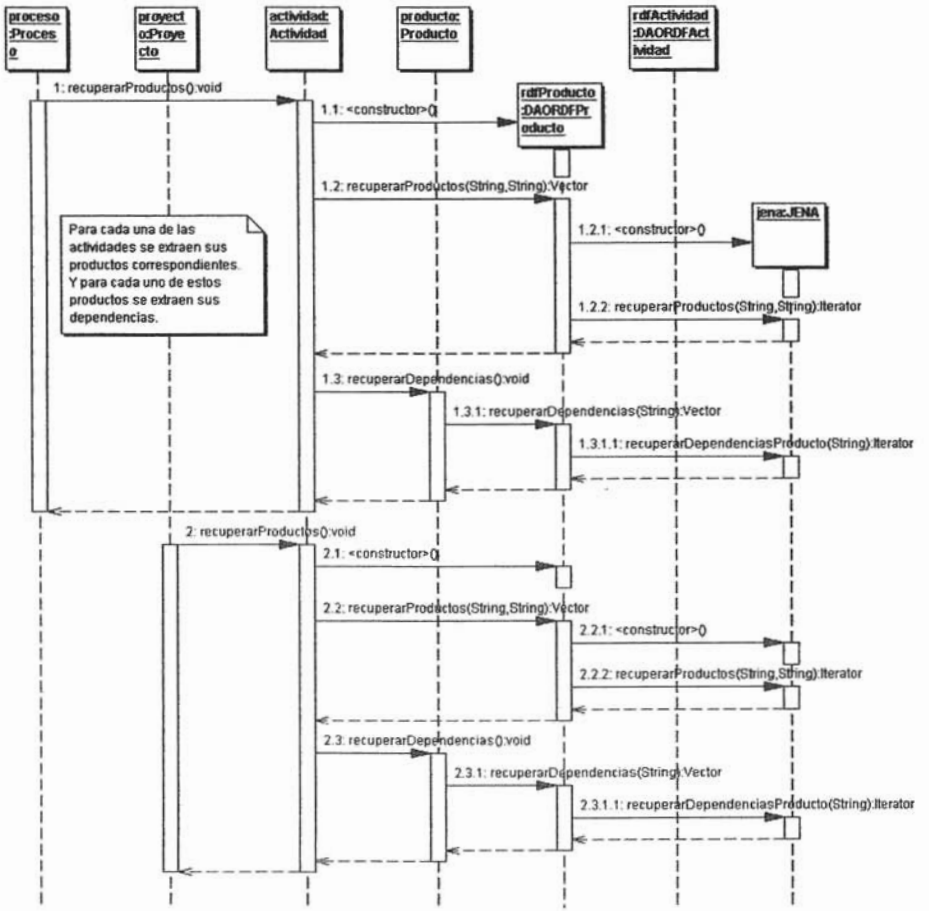

proceso :Proceso
proyecto:Proyecto
actividad: Actividad
producto: Producto
rdfActividad :DAORDFActividad
rdfProducto :DAORDFProducto
jena:JENA
1: recuperarProductos():void
1.1: <constructor>()
1.2: recuperarProductos(String,String):Vector
1.2.1: <constructor>()
1.2.2: recuperarProductos(String,String):Iterator
Para cada una de las actividades se extraen sus productos correspondientes. Y para cada uno de estos productos se extraen sus dependencias.
1.3: recuperarDependencias():void
1.3.1: recuperarDependencias(String):Vector
1.3.1.1: recuperarDependenciasProducto(String):Iterator
2: recuperarProductos():void
2.1: <constructor>()
2.2: recuperarProductos(String,String):Vector
2.2.1: <constructor>()
2.2.2: recuperarProductos(String,String):Iterator
2.3: recuperarDependencias():void
2.3.1: recuperarDependencias(String):Vector
2.3.1.1: recuperarDependenciasProducto(String):Iterator

**Figura 5. 15**

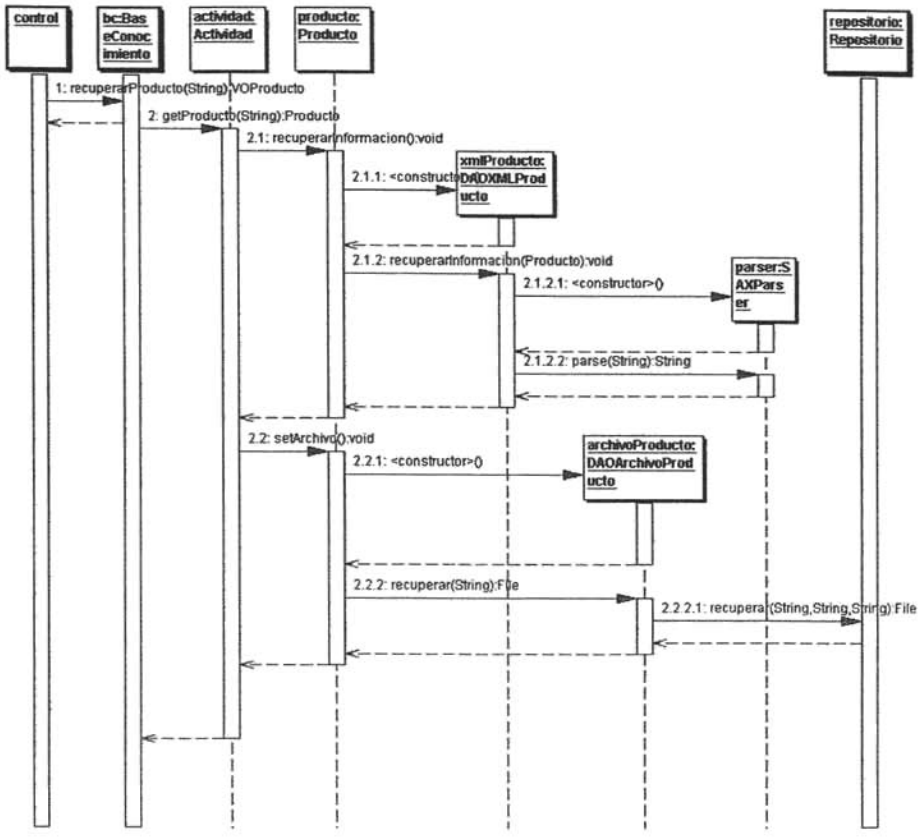



**Figura 5. 16**

### *Sincronizador*

El módulo sincronizador fue diseñado para garantizar la consistencia de los datos, provenientes de los distintos componentes de la persistencia. Este módulo, ver figura 5.17, es de uso interno de la BC y por lo tanto no tiene representación en su fachada.

La clase Sincronizador se encarga de mantener actualizados al Repositorio y al MarcoTrabajo, es decir, cualquier cambio en el repositorio es reflejado en el marco de trabajo, y viceversa. Ver el diagrama de secuencia (figura 5.18) para el caso de guardar un producto. La clase Sincronizador tiene la capacidad de registrar objetos *observer*, que a su vez guardan una referencia al sincronizador, permitiendo la comunicación en ambas direcciones. Cuando alguna modificación debe ser notificada el *observer* invoca el método *update*(), informando al sincronizador que debe avisar del cambio a los demás *observers*, para que estos

actualicen su propio estado. Es necesario implementar un protocolo de comunicación entre los *observers* que indique qué ha sido modificado y qué hacer al respecto. También se debe evitar que el *observer* que notifica sea notificado de vuelta y se provoque un ciclo infinito.

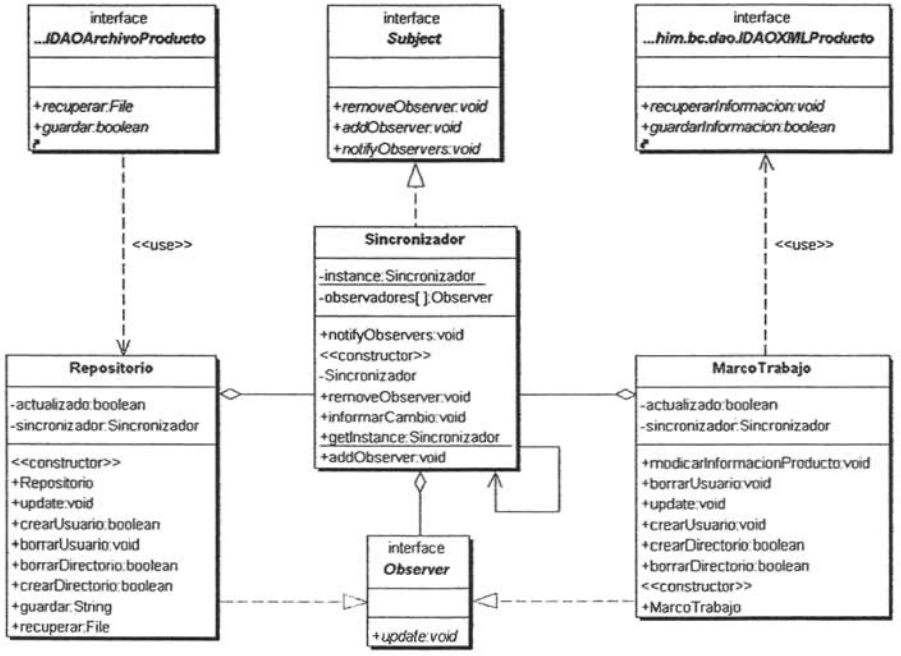



**Figura 5.17**

La clase Sincronizador implementa la interfaz *Subject*, y el Repositorio y el MarcoTrabajo implementan la interfaz *Observer*, formando al patrón del mismo nombre. Gracias a este esquema es posible añadir o sustituir los módulos que requieran ser sincronizados, imponiendo únicamente una clase que implemente la interfaz *Observer*. Esto permite, por ejemplo, que el marco de trabajo en lugar de ser manejado en RDF se mantenga en una base de datos relacional, sin tener que modificar el repositorio o el sincronizador.

Las clases Repositorio y MarcoTrabajo constituyen fachadas para los módulos de repositorio y RDF, respectivamente. Esta estructura facilita el acceso a la capa de persistencia desde la clase BaseConocimiento, evitando que se haga uso directo de las clases DAOs.

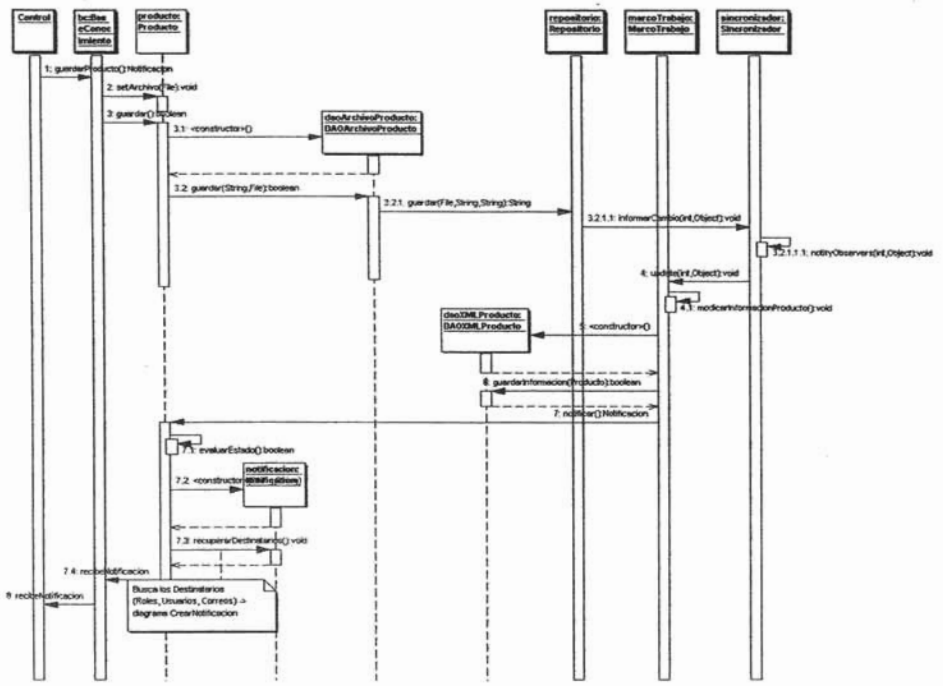



**Figura 5. 18**

## *Control*

La función básica del control es dar respuesta a las peticiones del usuario, proveyendo los recursos necesarios para realizar sus actividades. El control también es responsable de mantener consistente la aplicación, es decir, dependiendo de su estado y de una petición del usuario, el control debe ser capaz de decidir qué proceso realizar sin que la aplicación caiga en un estado inválido.

Las tareas que el usuario puede realizar son:

- Ingresar al sistema (*login*).
- Obtener los procesos y proyectos (esquemas) en los cuales está dado de alta, de acuerdo a los roles que tiene asociados.
- Elegir un esquema.
- Revisar las actividades que el esquema incluye.
- Realizar alguna actividad. La aplicación sólo habilita las actividades que el usuario puede llevar acabo de acuerdo al rol que eligió.
- Capturar un documento (PIE), o guardarlo (caja negra).

- Salir del sistema (*logoff*).

### *Ingresar al sistema*

En la pantalla de inicio se le presenta al usuario la opción de ingresar al sistema. Para realizar esta actividad se debe introducir el nombre de usuario y la contraseña. El sistema valida que hayan sido capturadas ambas cadenas, con ciertas restricciones como la longitud de las mismas. En caso que no se cumplan se envía un mensaje indicando el error.

Una vez que han sido validados los datos del usuario, el *actionServlet* elige, a través de un *actionMapping*, que *action* manejará la petición, en este caso es el *action*Seguridad que interacciona directamente con la Base de Conocimiento (BC) a través de la clase BaseConocimiento. Esta clase tiene los métodos necesarios para validar si el usuario está dado de alta, para recuperar su contraseña (*password*) y para obtener todos sus datos.

Si la validación fue exitosa el *action*Seguridad indica, a través de un *actionForward*, la pantalla que deberá mostrarse, que en este caso es el marco general. En caso contrario se genera un error que es encapsulado en un *action*Error, encargado de mostrar un mensaje descriptivo al usuario sobre la situación. En el diagrama de la figura 5.19 se muestra cómo están relacionados los componentes descritos:

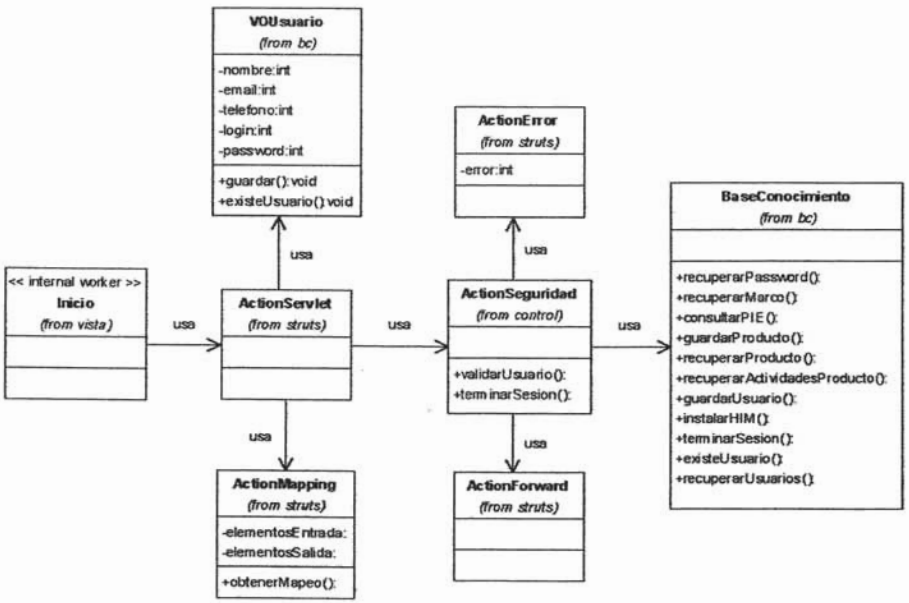



**Figura 5.19**

A continuación se muestran los diagramas de secuencia (figura 5.20 y 5.21) para la autenticación del usuario. La interacción mostrada incluye la capa de vista y control, y posteriormente la capa de control y el modelo.

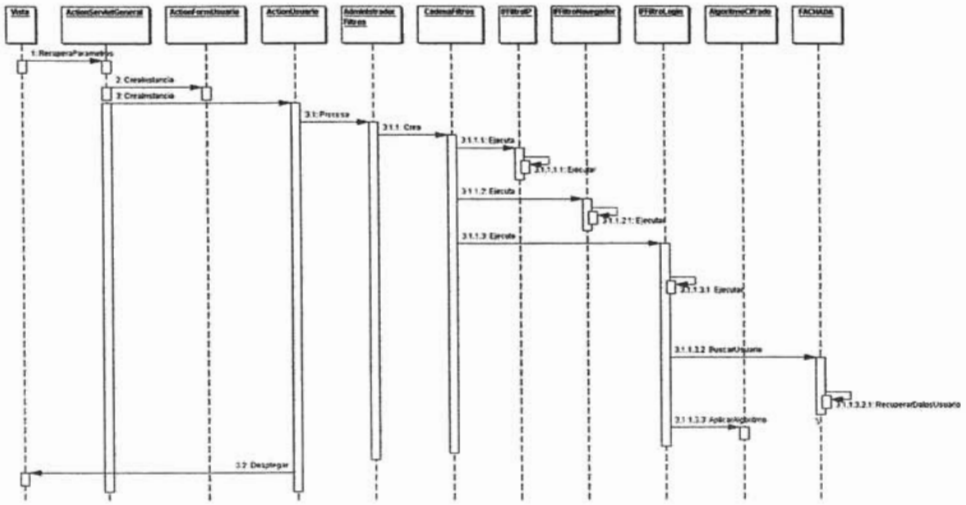



Figura 5. 20

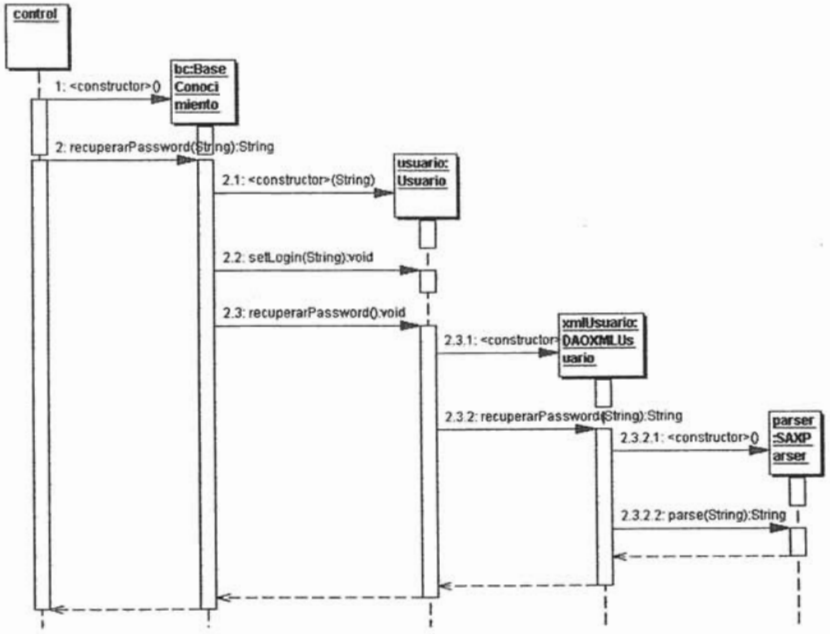



Figura 5. 21

Para aprovechar la funcionalidad que provee *Struts* sobre el llenado de *value objects* (VO) directamente de las formas de las pantallas, se hizo que los VO hereden de la clase *action*Form con lo cual se logra que carguen sus campos automáticamente, Esto se muestra en la figura 5.22.

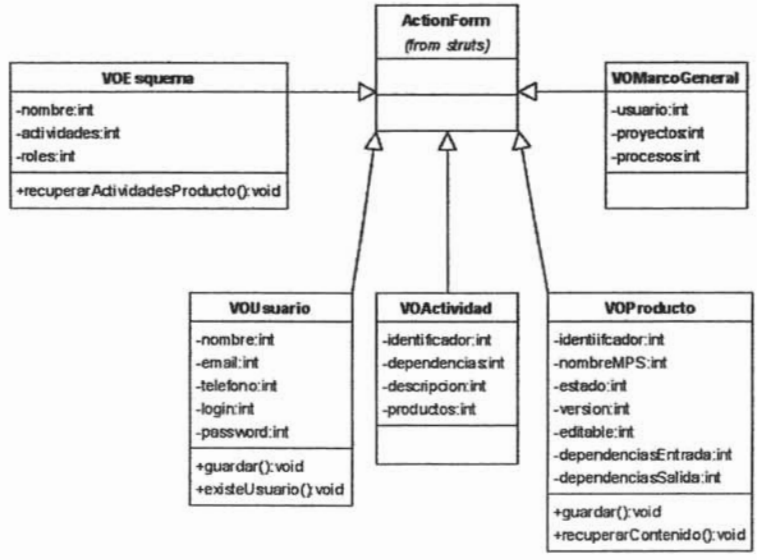



**Figura 5.22**

### *Presentar marco general*

Como ya se mencionó el marco general del usuario incluye los procesos y proyectos junto con sus roles. De esta manera se le presenta al usuario dos listas en las que podrá elegir el esquema que le interesa.

La forma de relacionar al usuario con un esquema es de acuerdo con los roles que incluye, y de la misma forma se recupera su marco de trabajo. Existe un usuario administrador que puede asociar inicialmente información a los usuarios. Posteriormente existen roles que se encargan del manejo de los recursos humanos y que podrán vincular a un usuario con roles específicos. El sistema almacena únicamente los roles inscritos y localiza los esquemas a los que pertenecen.

Este proceso constituye el primer nivel, y el más general, en la abstracción que el sistema representa de MoProSoft.

En el diagrama de la figura 5.23 se muestra cómo están relacionados los componentes descritos:

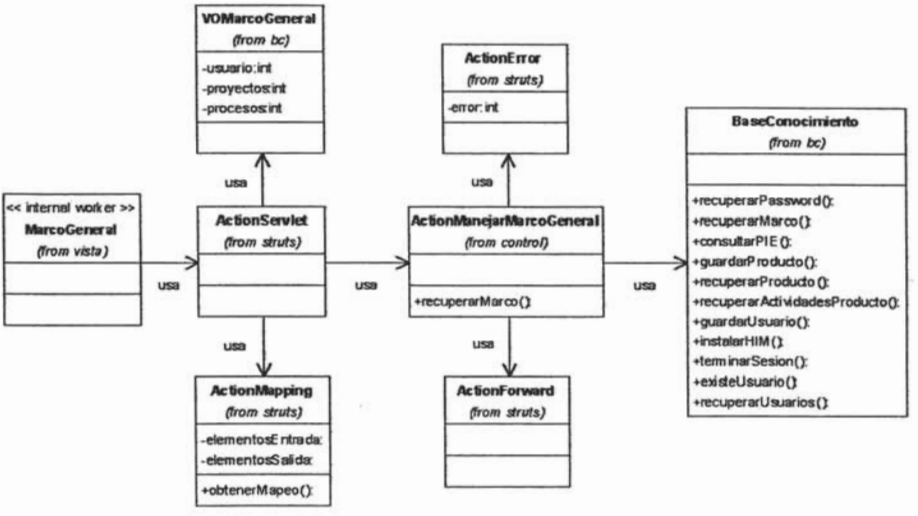



Figura 5.23

Una vez que el usuario eligió el esquema que desea, el sistema deberá proporcionar las actividades que lo conforman.

### *Realizar actividad*

Cuando el usuario llega a este nivel el sistema le muestra un árbol de actividades que representa la jerarquía de las mismas. El usuario puede consultar todas las actividades, pero sólo podrá realizar aquellas que estén asociadas a los roles que tiene. En cada caso se incluye una descripción, transcripción literal del documento de MoProSoft, también se adjunta la descripción de los productos que deberán ser generados como parte de la actividad, junto con sus dependencias.

Una vez que ha sido realizada una actividad, se registra la información pertinente en RDF y se actualiza el árbol de actividades para reflejar el cambio. Ver figura 5.24.

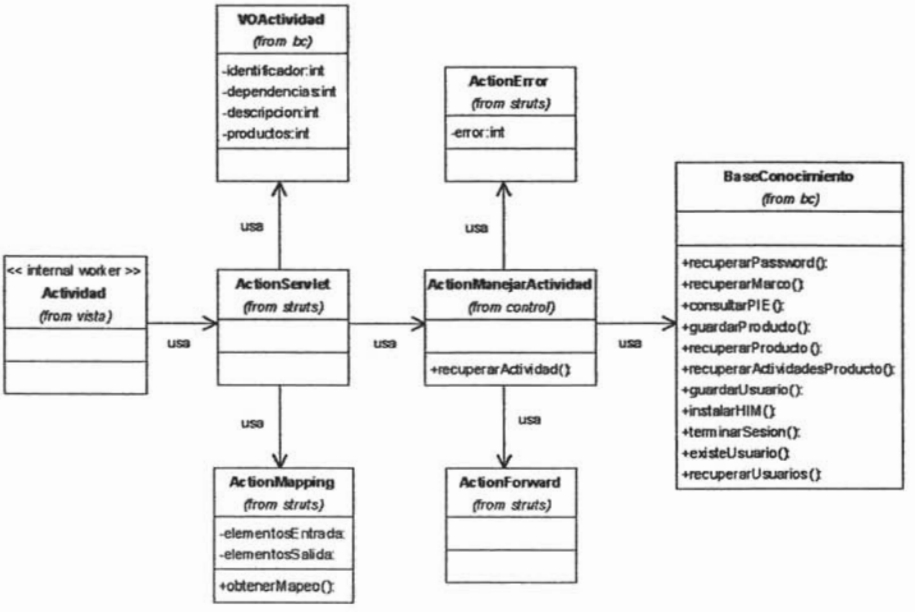



Figura 5.24

### *Producir documento*

Muchas de las actividades requieren que se produzca un documento. El sistema maneja dos tipos de documentos: los PIEs y las cajas negras (ver figura 5.25). Para los primeros el sistema proporciona formas de captura, mientras que los segundos son añadidos como archivos que son almacenados en el repositorio.

Comúnmente al registrar un documento el sistema marca como realizada la actividad. En algunos casos es necesario que el documento sea validado por otro rol. Para lograr esto el sistema genera y envía una notificación al usuario que tenga ese rol, en la que se especifica la situación del documento. Una vez que el documento es validado, y no se requieren cambios, se establece como terminado el producto; en caso contrario se repite el proceso.

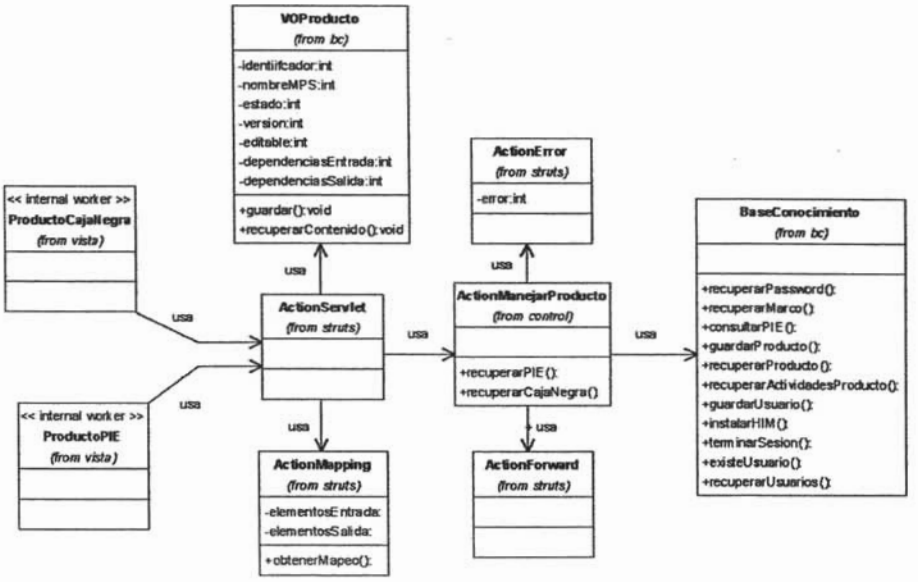



Figura 5.25

### *Salir del sistema*

Cuando el usuario elige salir del sistema se cierra su sesión y se liberan los recursos asociados.

El contenido de esta sección constituye una propuesta con un nivel de abstracción alto para el módulo de control, para un mayor detalle referirse a [Vázquez].

## *Vista*

En este módulo se incluyen todos los elementos necesarios para mostrar la información al usuario: *JSPs*, páginas HTML, bibliotecas de etiquetas personalizadas, clases Java, etc.

La pantalla del usuario está constituida por tres frames:

- *Encabezado*. Muestra el nombre de la aplicación, la fecha, la secuencia de páginas visitadas, etc.
- *Menú*. Las opciones que presenta el menú son contextuales. Es decir, dado un usuario sólo se habilitan las actividades que puede realizar de acuerdo al estado de la aplicación.

- *Área de trabajo*. En este frame se presentan los componentes que el usuario requiere para realizar una tarea, que incluyen campos de captura, textos de descripción, etc.

### *Componentes*

En la vista existen componentes para cada una de las tareas principales del usuario, como pueden ser entrar al sistema, elegir un esquema o capturar un producto. Estos componentes (*helpers*) implementan el patrón *View Helper*[17], y son alimentados con datos, que generalmente provienen de *value objects*, pero pueden residir en la sesión o ser enviados desde una página. Una vez que se reciben los datos los *helpers* se encargan de estructurarlos para generar la página requerida.

Todos los *helpers* implementan la clase FrameTrabajo, que por razones obvias constituye el componente que presenta el contenido del área de trabajo al usuario. Para facilitar la implementación de tareas comunes como estructurar una página HTML o extraer los valores de un *bean*, se hace uso de las bibliotecas de etiquetas personalizadas: *Struts*-Html, *Struts-Bean* y *Struts-Logic*[18].

Cada uno de los tres frames, mencionados anteriormente, es generado a partir de un *helper*. Este esquema de trabajo implementa el patrón *Composite View* y permite armar una pantalla dinámicamente a partir de componentes, en lugar de construir una pantalla por cada contexto que se requiera.

El diagrama de la figura 5.26 muestra los componentes de la vista.

[17] Para mayor referencia ver la sección *Vista* del capitulo Tecnologías y Estrategias para Patrones.
[18] Ver la sección *Arquitectura de Struts* del capitulo Tecnologia para Patrones.

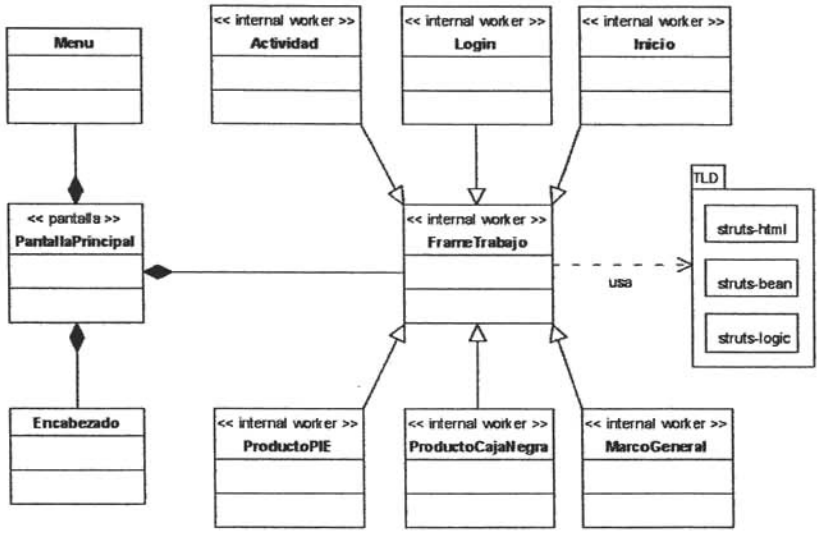



**Figura 5.26**

El contenido de esta sección constituye una propuesta con un nivel de abstracción alto para el módulo de vista, para un mayor detalle referirse a [Valdivieso].

# *Conclusiones*

## *¿Qué se hizo?*

El documento de la tesis debe constituir un ejemplo de la aplicación de buenas prácticas en la industria del software, de las cuales son incluidas: el uso de patrones, la aplicación de procesos basados en un modelo (MoProSoft) y el desarrollo apoyado en el Proceso Unificado.

Como se menciona anteriormente el objetivo de esta tesis consiste en especificar la arquitectura de HIM. Para lograrlo fue necesario establecer las bases sobre las que se construiría. Primeramente se determinó qué funcionalidad debería proveer la herramienta y se introdujeron los productos obtenidos a partir del análisis. Después fue incluida la definición de lo que se debe entender por un patrón de diseño y de arquitectura. Una vez concretados estos elementos se eligió el patrón que utlizaría la arquitectura: MVC.

Posteriormente se detallaron los patrones que la arquitectura incluye en cada uno de sus módulos y se proporcionaron las tecnologías para lograr la implementación. Estas tecnologías están apoyadas en componentes y bibliotecas que constituyen sistemas por si mismos, cuya complejidad y diversidad dificulta la tarea de elegir cuál es más apropiada para una funcionalidad específica. Para resolver esta situación se estableció un comparativo mostrando las ventajas y las desventajas que tiene cada tecnología al ser aplicada. En cada caso se consideraron características como la facilidad al implementar una funcionalidad, el permitir una clara separación entre capas, el estar incluidas en un framework, y en general el permitir la reutilización, la extensibilidad, la portabilidad y la escalabilidad.

En la siguiente tabla se muestran las estrategias elegidas para cada uno de los patrones.

| Patrón | Estrategias | Observaciones |
|---|---|---|
| ***Vista*** | | |
| *View Helper* | ✓ *JavaBean*<br>✓ *Custom Tag*<br>✓ *BusinessDelegate* | *Struts* usa las dos primeras estrategias en sus clases *ActionForm* y *DynaActionForm*[19].<br>La clase *Action* de *Struts* es un *BusinessDelegate* al ocultar los detalles del manejo de las solicitudes. |
| *Composite View* | ✓ *Custom Tag* | *Struts* incluye una biblioteca de etiquetas y componentes *Tiles*[20] para el manejo dinámico de los componentes de las pantallas. |

[19] Ver [Husted], página 160, "*ActionForm flavors*".
[20] Ver [Husted], página 319, "*Developing Applications with Tiles*".

| **Control** | | |
|---|---|---|
| *Front Controller* | ✓ *Servlet*<br>✓ *Command*<br>✓ *Logical Resouce Mapping* | La clase *ActionServlet* de *Struts* es el único punto de entrada al control desde la vista. Las clases *Action* son usadas como objetos *command* por el *ActionServlet* para delegar responsabilidades al responder las peticiones. El mapeo de las peticiones a los *Actions* es especificado a nombres lógicos en archivos XML. |
| *Dispatcher View* | ✓ *Controller* | La clase *ActionForward* de *Struts* se encarga de manejar la navegabilidad de los elementos de la vista y del control. |
| *Command* | ✓ *JavaBean* | Los objetos *Action* se encargan de procesar las solicitudes para lo cual pueden hacer uso de recursos del modelo. |
| **Modelo** | | |
| *Data Access Object* | ✓ *Regular* Java *Object* | Para implementar los objetos DAO se usaron interfaces que permitieran desacoplar los DAOs de las clases que los utilizan. Debido al alcance del proyecto no se definió un *DAOFactory*, el cual podría ser útil en caso de manejar la persistencia en más formatos de los incluidos (RDF y archivos). |
| *Observer*<br>*Session Façade*<br>*Business Object* | ✓ *Regular* Java *Object* | Debido al alcance del proyecto y a la complejidad que conllevan no se usaron EJBs (*session beans* o *message driven* beans). En el caso de la fachada es utilizó la estrategia *stateless*, al no mantenerse el estado entre cada invocación de sus métodos. Para proveer el acceso concurrente puede optarse por los EJBs, ya que los servidores de aplicación en donde residen incluyen esa funcionalidad. |
| *Value Object* (VO) | ✓ *Updateable*<br>✓ *Multiple* | Los *value objects* son usados para proveer y recibir datos, por lo tanto deben poder ser modificados (*updateable*). La segunda estrategia se usa en la clase VOMarcoGeneral, un VO que contiene procesos y proyectos que a su vez son VOs. |

En las capas de control y vista se eligió el *framework Struts*, y se proporcionó una descripción a profundidad del mismo, con lo que se busca aprovechar mejor los recursos que provee.

Una vez asentados los conceptos anteriores se procedió a definir la arquitectura de la herramienta, incluyendo cada uno de los componentes del patrón MVC, logrando el objetivo principal de la tesis. Para conseguir esto se usaron los diagramas de clases del diseño obtenidos durante el segundo ciclo del

desarrollo de HIM. En estos diagramas se aplican los patrones y las tecnologías planteadas. Es importante señalar que cada uno de los componentes de la arquitectura es tratado a profundidad en las otras cinco tesis[21] por lo cual no se hace aquí.

A fin de garantizar que fueron cumplidos los requerimientos iniciales se incluye un registro de rastreo de los casos de uso del negocio[22] indicando cuales fueron incluidos en HIM como casos de uso de la herramienta y posteriormente como clases del análisis y del diseño.

## *Trabajo futuro*

Debido a la magnitud del proyecto hubo funcionalidad que se dejó para trabajos futuros. Teniendo esto en mente, el esquema propuesto considera la inclusión de nueva funcionalidad, sin que esto implique mayores cambios. A continuación se listan los puntos de funcionalidad a incluir:

- *Notificar el cambio de estado de los productos.* Consiste en enviar correos electrónicos, u otro tipo de mensaje, a los usuarios que por sus roles deban conocer el cambio de estado de un producto. Generalmente los roles notificados serán el responsable del documento y los encargados de verificarlo y validarlo.
- *Manejo de repositorios personales.* Los documentos que son resguardados en el repositorio de HIM deben estar terminados, por lo cual mientras se está trabajando en ellos se deberán almacenar en la máquina del usuario. Para evitar la perdida de datos y mejorar el acceso a la herramienta desde cualquier computadora, se propone incluir repositorios donde los usuarios puedan almacenar sus documentos.
- *Administración de respaldos.* Incluye la automatización de los respaldos, la elección de la información que se desee respaldar, etc.
- *Soporte para el acceso concurrente.* Actualmente el sistema está estructurado para ser usado por un usuario, pero como se especifica en los requerimientos no funcionales[23] se deberá permitir el acceso por varios usuario de forma concurrente.
- *Búsquedas para explotar la información de los PIEs.* Para que la base de conocimiento sea considerada como tal, es necesario que la herramienta permita la consulta de los documentos almacenados (PIEs) basándose en criterios elegidos por el usuario.

[21] Ver [Hernández], [Mercado], [Valdivieso], [Vázquez] y [Cruz].
[22] Ver el apéndice.
[23] Ver el documento Requerimientos no Funcionales del primer ciclo incluido en el CD de HIM.

- *Manejo de documentos del tipo "caja negra"*. Como se ha comentado anteriormente, los documentos caja negra no podrán ser editados en la herramienta, pero si se deberá permitir indicar el programa que maneja cada uno de ellos, de manera que al solicitarlo sea desplegado adecuadamente.

# *Apéndice*

## *Registro de Rastreo*

En este apéndice se revisa la documentación generada en los ciclos I y II del proyecto HIM, con el fin de mostrar el mapeo de los requerimientos a través de las distintas fases del Proceso de Desarrollo. Se ha utilizado la noción de caso de uso (CU-N, en forma tabular) para modelar los aspectos de MoProSoft (Procesos y Actividades), posteriormente se han relacionado con los casos de uso de la herramienta (CU-H) necesarios para soportarlos. En la tercera columna de las tablas se muestran las Clases del Análisis (CA) consideradas para realizar los CU. Al final del documento se tiene una tabla con las Clases del Diseño que implementan los atributos y las responsabilidades, del modelado orientado a objetos, que las CA identifican.

*Gestión de Negocio*

| Casos de Uso del Negocio | Casos de Uso de la Herramienta | Clases del Análisis [0] |
|---|---|---|
| **A1. Planeación Estratégica** | | |
| A1.1, A1.3 - A1.10, A1.12 y A1.14 Generar / Corregir Plan Estratégico. | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A1.2 Entender la situación actual | Fuera del alcance del proyecto | |
| A1.11 Verificar el Plan Estratégico | A2.4.4 Verificar / Validar Producto | Proceso<br>Producto<br>Notificacion |
| A1.13 Validar el Plan Estratégico | | |
| A1.15 Generar Plan de Adquisiciones y Capacitación | A2.4 Generar / Modificar Producto | |
| **A2. Preparación para la Realización** | | |
| A2.1 Preparar el ambiente para la implantación del Plan Estratégico | Fuera del alcance del proyecto | |
| A2.2 y A2.4 Generar / Corregir Plan de Comunicación e Implantación | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A2.3 Validar el Plan de Comunicación e Implantación | A2.4.4 Verificar / Validar Producto | |

| A3. Valoración y Mejora Continua | | |
|---|---|---|
| A3.1 Analizar la información y evaluar el desempeño | A2.4.2 Editar / Consultar Producto | Proceso<br>Producto |
| A3.2 Generar Reporte de Valoración | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificación |
| A3.3 y A3.5 Generar / Corregir Propuesta de Mejoras al Plan Estratégico | | |
| A3.4 Validar la Propuesta de Mejoras | A2.4.4 Verificar / Validar Producto | |
| A3.6 Generar Reporte de Mediciones y Sugerencias de Mejora | A2.4 Generar / Modificar Producto | |
| A3.7 Identificar las Lecciones Aprendidas e Integrarlas a la BC | A2.4.5 Manejo de PIE | Proceso<br>PIE<br>Notificación |

*Gestión de Procesos*

| Casos de Uso del Negocio | Casos de Uso de la Herramienta | Clases del Análisis [9] |
|---|---|---|
| **A1. Planeación** | | |
| A1.1 - A1.7, A1.9 y A1.11<br>Generar / Corregir Plan de Procesos | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A1.8 Verificar el Plan Procesos | A2.4.4 Verificar / Validar Producto | |
| A1.10 Validar el Plan Procesos | | |
| **A2. Preparación para la Implantación** | | |
| A2.1 Gestionar el Plan de Adquisiciones y Capacitación | No soportado por la herramienta | |
| A2.2 Asignar y notificar a los Responsables de Procesos | A2.4.5 Manejo de PIE | Proceso<br>PIE<br>Notificacion |
| A2.3 y A2.5 Elaborar / Corregir Documentación de Procesos | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A2.4 Verificar la Documentación de Procesos | A2.4.4 Verificar / Validar Producto | |
| A2.6 Capacitar a la organización en los procesos | No soportado por la herramienta | |
| A2.7 Implantar los procesos en proyectos pilotos | | |

| Casos de Uso del Negocio | Casos de Uso de la Herramienta | Clases del Análisis [9] |
|---|---|---|
| **A3. Evaluación y control** | | |
| A3.1 Dar seguimiento a las actividades de implantación de procesos del Plan de Procesos | No soportado por la herramienta | |
| A3.2 Generar Reporte de Mediciones y Sugerencias de Mejora | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A3.3 Generar el Reporte Cuantitativo y Cualitativo | | |
| A3.4, A3.6 y A3.8 Generar / Corregir Plan de Acciones y Reporte de Evaluación | | |
| A3.5 Verificar el Plan de Acciones | A2.4.4 Verificar / Validar Producto | |
| A3.7 Validar el Plan de Acciones | | |
| A3.9, A3.11 y A3.13 Generar / Corregir el Plan de Mejora | A2.4 Generar / Modificar Producto | |
| A3.10 Verificar el Plan de Mejora | A2.4.4 Verificar / Validar Producto | |
| A3.12 Validar el Plan de Mejora | | |
| A3.14 Dar Seguimiento al Plan de Acciones y al Plan de Mejora | No soportado por la herramienta | |
| A3.15 Supervisar el control de riesgos | | |
| A3.16 Identificar las Lecciones Aprendidas e integrarlas a la BC | A2.4.5 Manejo de PIE | Proceso<br>PIE<br>Notificacion |

*Gestión de Proyectos*

| Casos de Uso del Negocio | Casos de Uso de la Herramienta | Clases del Análisis [9] |
|---|---|---|
| **A1. Planeación** | | |
| A1.1 Generar Alternativas de Realización de Proyectos Internos | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A1.2 Seleccionar una alternativa para los proyectos internos | No soportado por la herramienta | |
| A1.3 y A1.7 Generar / Corregir el Plan de Gestión de Proyectos | A2.4 Generar / Modificar Producto | Proceso<br>Producto |

| | | |
|---|---|---|
| A1.4 y A1.7 Generar / Corregir el Plan de Adquisiciones y Capacitación | | Notificacion |
| A1.5 y A1.7 Generar / Corregir Mecanismos de Comunicación con los Clientes | | |
| A1.6 Validar los tres documentos anteriores | A2.4.4 Verificar / Validar Producto | |
| **A2. Realización** | | |
| A2.1 Realizar actividades del Plan de Ventas | No soportado por la herramienta | |
| A2.2 Realizar actividades del Plan de Proyectos | | |
| A2.2.1 Generar Registro de Proyecto | A2.4.5 Manejo de PIE | Proceso<br>PIE<br>Notificacion |
| A2.2.2 Descripción del Proyecto | | |
| A2.2.3 Generar Metas Cuantitativas para el Proyecto | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A2.2.4 Asignar Responsable de Administración de Proyecto Específico | A2.4.5 Manejo de PIE | Proceso<br>PIE<br>Notificacion |
| A2.2.5 Recibir y aprobar el Plan del Proyecto | A2.4.3.4 Establecer estado del Producto | Proceso<br>Producto<br>Notificacion |
| A2.2.6 Recolectar los Reportes de Seguimiento | No soportado por la herramienta | |
| A2.2.7 Cerrar proyectos al recibir Documento de Aceptación | A2.4.3.4 Establecer estado del Producto | Proceso<br>Producto<br>Notificacion |
| A2.3 Implantar los Mecanismos de Comunicación con los Clientes | No soportado por la herramienta | |
| **A3. Evaluación y Control** | | |
| A3.1 Analizar el cumplimiento del Plan de Ventas | No soportado por la herramienta | |
| A3.2, A3.3, A3.5 Generar Acciones Correctivas o Preventivas (Reportes de Seguimiento, Quejas del Cliente) | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |

| | | |
|---|---|---|
| A3.4 Generar el Reporte Cuantitativo y Cualitativo | | |
| A3.6 Generar el Reporte de Mediciones y Sugerencias de Mejora | | |
| A3.7 Identificar las Lecciones Aprendidas e integrarlas a la BC | A2.4.5 Manejo de PIE | Proceso<br>PIE<br>Notificacion |

*Gestión de Recursos*

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| **A1. Planeación de Recursos** | | |
| A1.1 Generar el Plan de Adquisiciones y Capacitación | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A1.2 y A1.4 Generar / Corregir el Plan Operativo de Recursos Humanos y Ambiente de Trabajo | | |
| A1.3 Verificar el Plan Operativo de Recursos Humanos y Ambiente de Trabajo | A2.4.4 Verificar / Validar Producto | |
| A1.5 y A1.7 Generar / Corregir el Plan Operativo de Bienes, Servicios e Infraestructura | A2.4 Generar / Modificar Producto | |
| A1.6 Verificar el Plan Operativo de Bienes, Servicios e Infraestructura | A2.4.4 Verificar / Validar Producto | |
| A1.8 y A1.10 Generar / Corregir el Plan Operativo de Conocimiento de la Organización | A2.4 Generar / Modificar Producto | |
| A1.9 Verificar el Plan Operativo de Conocimiento de la Organización | A2.4.4 Verificar / Validar Producto | |
| **A2. Seguimiento y Control** | | |
| A2.1 Dar seguimiento a la ejecución del Plan Operativo de Recursos Humanos y Ambiente de Trabajo | Fuera del alcance del proyecto | |
| A2.2 Dar seguimiento a la ejecución del Plan Operativo de Bienes, Servicios e Infraestructura | | |
| A2.3 Dar seguimiento a la ejecución del Plan Operativo de Conocimiento de la Organización | | |

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| A2.1.3, A2.2.3 y A2.3.3 Generar Acciones Correctivas en caso de detectar desviación en la ejecución de los planes respectivos | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A2.4 Generar el Reporte Cuantitativo y Cualitativo | | |
| A2.5 Generar el Reporte de Mediciones y Sugerencias de Mejora | | |
| A2.6 Identificar las Lecciones Aprendidas e integrarlas a la BC | A2.4.5 Manejo de PIE | Proceso<br>PIE<br>Notificación |
| **A3. Evaluación y Control** | | |
| A3.1 Generar Propuestas Tecnológicas | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |

*Recursos Humanos y Ambiente de Trabajo*

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| **A1. Preparación** | | |
| A1.1 Revisión del Plan Operativo de Recursos Humanos y Ambiente de Trabajo | A2.4.2 Editar / Consultar Producto | Proceso<br>Producto |
| A1.2 Definir criterios para recursos, capacitación, etc. | Fuera del alcance del proyecto | |
| A1.3 y A1.5 Generar / Corregir el Plan de Capacitación | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A1.4 Validar el Plan de Capacitación | A2.4.4 Verificar / Validar Producto | |
| A1.6 y A1.8 Generar / Corregir la Evaluación de Desempeño | A2.4 Generar / Modificar Producto | |
| A1.7 Validar la Evaluación de Desempeño | A2.4.4 Verificar / Validar Producto | |
| A1.9 y A1.11 Generar / Corregir la Encuesta sobre el Ambiente de Trabajo | A2.4 Generar / Modificar Producto | |
| A1.10 Validar la Encuesta sobre el Ambiente de Trabajo | A2.4.4 Verificar / Validar Producto | |

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| **A2. Instrumentación** | | |
| A2.1 Seleccionar, asignar y obtener la aceptación de los recursos humanos | | |
| A2.1.1 Seleccionar recurso humano en función al perfil solicitado | A2.4.5 Manejo de PIE (Registro de Recursos Humanos) | Proceso<br>PIE<br>Notificación |
| A2.1.2 Generar la Asignación de Recursos y notificar al solicitante | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A2.1.3 y A2.1.4 Obtener la aceptación de la Asignación de Recursos; en caso de rechazo se repite. | A2.4.3.4 Establecer estado del Producto | |
| A2.1.5 En caso de nuevo personal, registrar en Registro de Recursos Humanos | A2.4.5 Manejo de PIE | Proceso<br>PIE<br>Notificación |
| A2.2 Llevar a cabo el Plan de Capacitación | | |
| A2.2.1 Elaborar el Plan de Capacitación | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A2.2.2 Registrar la capacitación en el Registro de Recursos Humanos | A2.4.5 Manejo de PIE | Proceso<br>PIE<br>Notificación |
| A2.3 Registrar el resultado de la Evaluación de Desempeño en Registro de Recursos Humanos | | |
| A2.4 Generar el Reporte de Ambiente de Trabajo | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| **A3. Generación de Reportes** | | |
| A3.1 Generar el Reporte de Recursos Humanos Disponibles, Capacitación y Ambiente de Trabajo | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A3.2 Generar el Reporte de Mediciones y Sugerencias de Mejora | | |

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| A3.3 Identificar las Lecciones Aprendidas e integrarlas a la BC | A2.4.5 Manejo de PIE | Proceso<br>PIE<br>Notificacion |

*Bienes, Servicios e Infraestructura*

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| **A1. Preparación** | | |
| A1.1 Revisión del Plan Operativo de Bienes, Servicios e Infraestructura | A2.4.2 Editar / Consultar Producto | Proceso<br>Producto |
| A1.2 Definir criterios para recursos, capacitación, etc. | Fuera del alcance del proyecto | |
| A1.3 y A1.5 Generar / Corregir el Plan de Mantenimiento | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A1.4 Validar el Plan de Mantenimiento | A2.4.4 Verificar / Validar Producto | |
| A1.6 Generar la Solicitud de Bienes o Servicios | A2.4 Generar / Modificar Producto | |
| **A2. Instrumentación** | | |
| A2.1 Adquirir el bien o servicio | | |
| A2.1.1 Seleccionar los proveedores del Catálogo de Proveedores | A2.4.5 Manejo de PIE | Proceso<br>PIE<br>Notificacion |
| A2.1.2 Obtener los presupuestos y descripción del bien o servicio | Fuera del alcance del proyecto | |
| A2.1.3 Pedir la selección del proveedor | | |
| A2.1.4 y A2.1.5 Adquirir el bien o servicio y pedir su aceptación. En caso de rechazo se repite. | | |
| A2.1.6 Registrar el bien o servicio aceptado en el Registro de Bienes o Servicios | A2.4.5 Manejo de PIE | Proceso<br>PIE<br>Notificacion |
| A2.1.7 En caso de adquirir el bien o servicio de un proveedor nuevo, registrarlo en el Catálogo de Proveedores (CP) | | |
| A2.1.8 Evaluar la satisfacción del solicitante por el bien o servicio adquirido, y registrarla en el CP | | |

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| A2.2 Crear el Registro de Mantenimiento | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| **A3. Generación de Reportes** | | |
| A3.1 Generar el Reporte de Bienes, Servicios e Infraestructura | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A3.2 Generar el Reporte de Mediciones y Sugerencias de Mejora | | |
| A3.3 Identificar las Lecciones Aprendidas e integrarlas a la BC | A2.4.5 Manejo de PIE | Proceso<br>PIE<br>Notificacion |

*Conocimiento de la Organización*

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| **A1. Planeación** | | |
| A1.1 – A1.4 y A1.6 Generar / Corregir el Plan de Administración de la BC | A2.4.2 Editar / Consultar Producto | Proceso<br>Producto<br>Notificacion |
| A1.5 Validar el Plan de Administración de la BC | A2.4.4 Verificar / Validar Producto | |
| **A2. Realización** | | |
| A2.1 – A2.3 y A2.5 Generar / Corregir el Diseño de la BC | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A2.4 Validar el Diseño de la BC | A2.4.4 Verificar / Validar Producto | |
| A2.6 Poner en operación y dar mantenimiento a la BC | A1.1 Generar BC | BaseConocimiento |
| **A3. Evaluación y Control** | | |
| A3.1 Revisar el uso de la BC | A2.4.2 Editar / Consultar Producto | Proceso<br>Producto<br>BitacoraSistema |
| A3.2 Generar un Reporte del Estado de la BC | A2.4 Generar / Modificar Producto | Proceso<br>Producto<br>Notificacion |
| A3.3 Generar el Reporte de Mediciones y Sugerencias de Mejora | | |

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| A3.4 Identificar las Lecciones Aprendidas e integrarlas a la BC | A2.4.5 Manejo de PIE | Proceso<br>PIE<br>Notificacion |

*Administración de Proyectos Específicos*

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| **A1. Planeación** | | |
| A1.1 Revisar la Descripción del Proyecto | A2.4.2 Editar / Consultar Producto | Proceso y Proyecto<br>Producto<br>Notificacion |
| A1.2 Generar el Proceso Específico del proyecto | A2.4 Generar / Modificar Producto | |
| A1.3 Generar el Protocolo de Entrega | | |
| A1.4 Generar Ciclos y Actividades | | |
| A1.5 Generar la relación y dependencia de cada actividad | | |
| A1.6 Generar el Tiempo Estimado para cada actividad. | | |
| A1.7 Generar el Plan de Adquisiciones y Capacitación | | |
| A1.8 Conformar el Equipo de trabajo | A2.4.5 Manejo de PIE | Proceso y Proyecto<br>PIE<br>Notificacion |
| A1.9 Generar el Calendario de trabajo | A2.4 Generar / Modificar Producto | Proceso y Proyecto<br>Producto<br>Notificacion |
| A1.10 Generar el Costo Estimado del Proyecto | | |
| A1.11 Generar/Modificar el Plan de Manejo de Riesgos | | |
| A1.12, A1.15 y A1.17 Generar/Modificar el Plan del Proyecto | A2.4.5 Manejo de PIE | Proceso y Proyecto<br>PIE<br>Notificacion |
| A1.13, A1.15 y A1.17 Generar/Modificar el Plan de Desarrollo | | |
| A1.14 Verificar el Plan del Proyecto y el Plan de Desarrollo | A2.4.4 Verificar / Validar Producto | Proceso y Proyecto<br>Producto<br>Notificacion |
| A1.16 Validar el Plan del Proyecto y el Plan de Desarrollo | | |
| A1.18 Iniciar formalmente un nuevo ciclo | Fuera del alcance del proyecto | |

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| **A2. Realización** | | |
| A2.1 Acordar la asignación de tareas | A2.4.5 Manejo de PIE | Proceso y Proyecto<br>PIE<br>Notificacion |
| A2.2 Acordar la distribución de la información | Fuera del alcance del proyecto | |
| A2.3 Revisar la Descripción del Producto, el Equipo de Trabajo y el Calendario | A2.4.2 Editar / Consultar Producto | Proceso y Proyecto<br>Producto<br>Notificacion |
| A2.4 Dar seguimiento al Plan de Adquisiciones y Capacitación. Aceptar/Rechazar la Asignación de Recursos Humanos o subcontratistas. Distribuir los recursos a los miembros del equipo | Fuera del alcance del proyecto | |
| A2.5 Manejar la relación con subcontratistas | | |
| A2.6 Recolectar y analizar los Reportes de Actividades, Reportes de Mediciones y Sugerencias de mejora y productos de trabajo | A2.4.2 Editar / Consultar Producto | Proceso y Proyecto<br>Producto<br>Notificacion |
| A2.7 Registrar costos y recursos reales | A2.4 Generar / Modificar Producto | |
| A2.8 Revisar el Registro de Rastreo | A2.4.2 Editar / Consultar Producto | |
| A2.9 Revisar la Configuración de Software | | |
| A2.10 Modificar el Plan del Proyecto y el Plan de Desarrollo según las Solicitudes de Cambios | A2.4 Generar / Modificar Producto | |
| A2.11 Generar Minutas de las reuniones de revisión | | |
| **A3. Evaluación y Control** | | |
| A3.1 Generar Acciones Correctivas en base al cumplimiento del Plan del Proyecto y del Plan de Desarrollo | A2.4 Generar / Modificar Producto | Proceso y Proyecto<br>Producto<br>Notificacion |
| A3.2 Actualizar el Plan de Manejo de Riesgos | | |
| A3.3 Generar el Reporte de Seguimiento del proyecto | | |
| **A4. Cierre** | | |
| A4.1 Generar el Documento de Aceptación | A2.4 Generar / Modificar Producto | Proceso y Proyecto |

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| | | Producto<br>Notificacion |
| A4.2 Efectuar el cierre con subcontratistas | Fuera del alcance del proyecto | |
| A4.3 Generar el Reporte de Mediciones y Sugerencias de Mejora | A2.4 Generar / Modificar Producto | Proceso y Proyecto<br>Producto<br>Notificacion |
| A4.4 Identificar las Lecciones Aprendidas | A2.4.5 Manejo de PIE | Proceso y Proyecto<br>PIE<br>Notificacion |

*Desarrollo y Mantenimiento de Software*

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| **A1. Realización de la fase de Inicio** | | |
| A1.1 Revisar el Plan de Desarrollo. | A2.4.2 Editar / Consultar Producto | Proceso y Proyecto<br>Producto<br>Notificacion |
| A1.2 Generar el Reporte de Actividades. | A2.4 Generar / Modificar Producto | |
| **A2. Realización de la fase de Requerimientos** | | |
| A2.1 Distribuir tareas a los miembros del equipo | A2.4.5 Manejo de PIE | Proceso y Proyecto<br>PIE<br>Notificacion |
| A2.2, A2.4 y A2.6 Generar/Modificar la Especificación de Requerimientos. | A2.4 Generar / Modificar Producto | Proceso y Proyecto<br>Producto<br>Notificacion |
| A2.3 Verificar la Especificación de Requerimientos. | A2.4.4 Verificar / Validar Producto | |
| A2.5 Validar la Especificación de Requerimientos. | | |
| A2.7 y A2.9 Generar/Modificar el Plan de Pruebas del Sistema. | A2.4 Generar / Modificar Producto | |
| A2.8 Validar el Plan de Pruebas del Sistema. | A2.4.4 Verificar / Validar Producto | |
| A2.10 y A2.12 Generar / Modificar el Manual de Usuario. | A2.4 Generar / Modificar Producto | |
| A2.11 Verificar el Manual de Usuario | A2.4.4 Verificar / Validar Producto | |

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| A2.13 Establecer documentos como línea base a la Configuración de Software | A2.4.3.4 Establecer estado del Producto | |
| A2.14 Elaborar el Reporte de Actividades. | A2.4 Generar / Modificar Producto | |
| **A3. Realización de la fase de Análisis y Diseño** | | |
| A3.1 Distribuir tareas en el equipo de trabajo según roles | A2.4.5 Manejo de PIE | Proceso y Proyecto<br>PIE<br>Notificacion |
| A3.2, A3.4 y A3.6 Generar / Modificar el Análisis y Diseño, y el Registro de Rastreo | A2.4 Generar / Modificar Producto | Proceso y Proyecto<br>Producto<br>Notificacion |
| A3.3 Verificar el Análisis y Diseño, y el Registro de Rastreo | A2.4.4 Verificar / Validar Producto | |
| A3.5 Validar el Análisis y Diseño | | |
| A3.7 y A3.9 Generar / Modificar el Plan de Pruebas del Integración | A2.4 Generar / Modificar Producto | |
| A3.8 Verificar el Plan de Pruebas del Integración | A2.4.4 Verificar / Validar Producto | |
| A3.10 Establecer documentos como línea base a la Configuración de Software | A2.4.3.4 Establecer estado del Producto | |
| A3.11 Elaborar el Reporte de Actividades | A2.4 Generar / Modificar Producto | |
| **A4. Realización de la fase de Construcción** | | |
| A4.1 Distribuir tareas en el equipo de trabajo según roles | A2.4.5 Manejo de PIE | Proceso y Proyecto<br>PIE<br>Notificacion |
| A4.2 Construir o modificar los Componentes de software | Fuera del alcance del proyecto | |
| A4.2.1 – A4.2.3 Implementar y corregir los Componentes | | |
| A4.2.4 y A4.4 Actualizar / Corregir el Registro de Rastreo | A2.4 Generar / Modificar Producto | Proceso y Proyecto<br>Producto<br>Notificacion |
| A4.3 Verificar el Registro de Rastreo | A2.4.4 Verificar / Validar Producto | |
| A4.5 Establecer documentos como línea base a la Configuración de Software | A2.4.3.4 Establecer estado del Producto | |
| A4.6 Elaborar el Reporte de Actividades | A2.4 Generar / Modificar Producto | |

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| **A5. Realización de la fase de Integración y Pruebas** | | |
| A5.1 Distribuir tareas en el equipo de trabajo según roles | A2.4.5 Manejo de PIE | Proceso y Proyecto<br>PIE<br>Notificacion |
| A5.2 Realizar integración y pruebas | Fuera del alcance del proyecto | |
| A5.2.1 y A5.2.2 Integrar los componentes, aplicar Plan de pruebas de Integración y corregir en base al Reporte de Pruebas de Integración | | |
| A5.2.3 Actualizar el Registro de Rastreo | A2.4 Generar / Modificar Producto | Proceso y Proyecto<br>Producto<br>Notificacion |
| A5.3 y A5.5 Generar / Modificar el Manual de Operación | | |
| A.5.4 Verificar el Manual de Operación | A2.4.4 Verificar / Validar Producto | |
| A5.6 Generar el Reporte de Pruebas del Sistema | A2.4 Generar / Modificar Producto | |
| A5.7 Corregir defectos encontrados con base al Reporte de Pruebas de Sistema | Fuera del alcance del proyecto | |
| A5.8 y A5.10 Generar / Modificar el Manual de Usuario | A2.4 Generar / Modificar Producto | Proceso y Proyecto<br>Producto<br>Notificacion |
| A5.9 Verificar el Manual de Usuario | A2.4.4 Verificar / Validar Producto | |
| A5.11 Establecer documentos como línea base a la Configuración de Software | A2.4.3.4 Establecer estado del Producto | |
| A5.12 Elaborar el Reporte de Actividades | A2.4 Generar / Modificar Producto | |
| **A6. Realización de la Fase de Cierre** | | |
| A6.1 y A6.3 Generar / Modificar el Manual de Mantenimiento | A2.4 Generar / Modificar Producto | Proceso y Proyecto<br>Producto<br>Notificacion |
| A6.2 Verificar el Manual de Mantenimiento | A2.4.4 Verificar / Validar Producto | |
| A6.4 Establecer documentos como línea base a la Configuración de Software | A2.4.3.4 Establecer estado del Producto | |
| A6.5 Identificar las Lecciones Aprendidas e integrarlas a la BC | A2.4.5 Manejo de PIE | Proceso y Proyecto<br>PIE<br>Notificacion |
| A6.6 Generar el Reporte de Mediciones y Sugerencias de Mejora | A2.4 Generar / Modificar Producto | Proceso y Proyecto<br>Producto |

| Casos de uso del negocio | Casos de uso de la herramienta | Clases del Análisis |
|---|---|---|
| A6.7 Elaborar el Reporte de Actividades | | Notificacion |

*Mapeo de Clases del Análisis al Diseño*

| Clases del Análisis | Clases del Diseño | |
|---|---|---|
| BaseConocimiento | • BaseConocimiento<br>• VOMarcoGeneral<br>• VOEsquema | • VOUsuario<br>• VOProducto<br>• VOActividad |
| MarcoGeneral | • MarcoGeneral<br>• IDAORDFProceso | • IDAORDFProyecto<br>• Jena |
| Proyecto<br>Proceso | • Proyecto \| Proceso<br>• Esquema<br>• IDAORDFActividad | • IDAOXMLRol<br>• Jena<br>• SaxParser |
| Actividad | • Actividad<br>• IDAORDFActividad<br>• DependenciaValor | • IDAORDFProducto<br>• Jena |
| Producto | • Producto<br>• ProductoPIE<br>• ProductoCajaNegra<br>• IDAOXMLUsuario<br>• IDAOArchivoProducto | • IDAOXMLProducto<br>• IDAORDFProducto<br>• Notificación<br>• Jena<br>• SaxParser |
| Rol | • IDAOXMLRol | |
| Usuario | • Usuario<br>• IDAOXMLUsuario | • SaxParser |

| Clases del Análisis | Clases del Diseño | |
|---|---|---|
| Notificación | • Notificación | |
| PIE | • ProductoPIE<br>• DAOXMLPIE | • IDAOXMLPIE<br>• CaracteristicaValor |
| BaseConocimiento | • BaseConocimiento (Fachada)<br>• VOMarcoGeneral | • VOActividad<br>• VOProducto<br>• VOEsquema |
| BitacoraSistema | • | • |

El Plan de Pruebas de Integración se aplica a los métodos de la clase BaseConocimiento (Fachada), en su integración con la capa de control.

## *Observaciones*

Las siguientes relaciones entre CU-N y CU-H serán soportadas por las clases del análisis como se especifica en la tercera y cuarta columna.

| Proceso | Casos de Uso del Negocio | Casos de Uso de la Herramienta | Consideración | Clases del Análisis [9] |
|---|---|---|---|---|
| GPr | A2.2 Asignar y notificar a los Responsables de Procesos | A2.4.5 Manejo de PIE | Soportarlo como archivo de configuración, y registrarlo desde la interfaz | Proceso<br>PIE<br>Notificacion |
| GPy | A2.2.5 Recibir y aprobar el Plan del Proyecto | A2.4.3.4 Establecer estado del Producto | Establecer ***aprobado*** como un estado del documento | Producto<br>Notificacion |
| | A2.2.7 Cerrar proyectos al recibir Documento de Aceptación | | Establecer ***cerrado*** como un estado del documento | |

| Proceso | Casos de Uso del Negocio | Casos de Uso de la Herramienta | Consideración | Clases del Análisis [0] |
|---|---|---|---|---|
| RHAT | A2.1.3 y A2.1.4 Obtener la aceptación de la Asignación de Recursos; en caso de rechazo se repite | | Establecer ***aceptado*** como un estado del documento | |
| CO | A3.1 Revisar el uso de la BC | A2.4.2 Editar / Consultar Producto | Convertir la información manejada por la clase BitacoraSistema en un producto para ser mostrado en la interfaz de usuario | Proceso<br>Producto<br>BitacoraSistema |
| DM | A2.1 Distribuir tareas a los miembros del equipo.<br>(A3.1, A4.1, A5.1) | A2.4.5 Manejo de PIE | Registrar esta información en el Registro de Recursos Humanos y/o en el Plan de Desarrollo | Proceso-Proyecto<br>PIE<br>Notificacion |
| | A2.13 Establecer documentos como línea base a la Configuración de Software. (A3.10, A4.5, A5.11, A6.4) | A2.4.3.4 Establecer estado del Producto | Establecer ***lineaBase*** como un estado de los documentos y/o modificar Configuración de Software | Proceso-Proyecto<br>Producto<br>Notificacion |

[0] Todas las realizaciones de casos de uso incluyen las Clases del Análisis:

- Marco General
- Actividad
- Rol
- Usuario

# Bibliografía

[Alexander] The Timeless Way of Building
Alexander, Christopher.
Oxford University Press, 1979.

[Alur] Core J2EE Patterns: Best Practices and Design Strategies
Alur, Crupi y Malks
Prentice Hall / Sun Microsystems Press, 2003.

[Armstrong] The J2EE™ 1.4 Tutorial
Armstrong, Eric, et al.
Sun Microsystems, 2003

[Booch] El Lenguaje Unificado de Modelado
Booch, Grady; Rumbaugh, James y Jacobson, Ivar.
Pearson Educación, 1999.

[Buschmann] A System of Patterns
Buschmann, Frank, et al.
John Wiley & Sons.

[CDHIM] CD Rom con la documentación generada en el primer y segundo ciclo del desarrollo de HIM.
Grupo de ISOO, generación 2003, Posgrado en Ciencias e Ingeniería de la Computación de la UNAM.

[Cruz] Análisis e Implementación de Esquemas de Seguridad Aplicados a la Herramienta Integral MoProSoft (HIM). Tesis de Maestría en Ingeniería (Computación) del Posgrado en Ciencias e Ingeniería de la Computación de la UNAM.
Cruz Vázquez, Jorge

[Gamma] Design Patterns Elements of Reusable *Object* Oriented Software
Gamma, Helm, Johnson, Vlissides
Addison Wesley, 1998.

[Hernández] Uso de la tecnología RDF para representar y manejar los procesos MoProSoft y su aplicación en HIM. Tesis de Maestría en Ingeniería (Computación) del Posgrado en Ciencias e Ingeniería de la Computación de la UNAM.
Hernández Uribe, Ernesto

[Horstmann] Java 2. Volumen 1. Fundamentos
Horstmann, Cay S. y Cornell, Gary.
Pearson Educación, 2003.

[HP] Jena 2 - A Semantic Web *Framework*
http://www.hpl.hp.com/semweb/jena

[Husted] *Struts in action*
Husted et al.
Mannig, 2003.

[Jacobson] El Proceso Unificado de Desarrollo de Software
Jacobson, Ivar, Booch, Grady y Rumbaugh, James
Pearson Educación, 2000.

[Mercado] La Sincronización de los Elemento de una Base de Conocimiento para MoProSoft y su aplicación en una Herrmienta Integral. Tesis de Maestría en Ingeniería (Computación) del Posgrado en Ciencias e Ingeniería de la Computación de la UNAM.
Mercado Fernández, Araceli

[Oktaba 2] Modelo de Procesos para la Industria de Software (MoProSoft) v 1.1
Oktaba, Hanna, et al. 2003

[Powers] Practical RDF
Powers, Shelley
O'Reilly & Associates, 2003.

[Stelting] Patrones de diseño aplicados a Java
Stelting, Maassen
Pearson Educación, 2003

[Struts] *Struts* – Documentation
Documentación del *framework* Jakarta *Struts* versión 1.1
jakarta.apache.org/*Struts*/

[Sun – 1] J2EE Patterns Catalog
Sun Java Center J2EE Patterns
Versión 1.0 beta, 2000

[Sun – 2] Especificaciones de tecnologías de Java (*JSPs*, *Servlets*, EJB, Swing)
http://java.sun.com/reference/docs/index.html

[Valdivieso] Utilización de patrones y la arquitectura J2EE para el diseño de la interfaz de usuario de la herramienta Integral MoProSoft (HIM). Tesis de Maestría en Ingeniería (Computación) del Posgrado en Ciencias e Ingeniería de la Computación de la UNAM.
Valdivieso, Karín

[Vázquez] Aplicación de patrones basados en J2EE para el diseño e implementación de la capa de control de la Herramienta Integral para MoProSoft (HIM). Tesis de Maestría en Ingeniería (Computación) del Posgrado en Ciencias e Ingeniería de la Computación de la UNAM.
Vázquez Morales, Marcos Oscar

[W3C] Semantic Web
W3C
http://www.w3.org/2001/sw/

[Yang] E++: A pattern *Language* for J2EE applications
Yang, Bin.
www.javaworld.com/javaworld/jw-04-2001/jw-0420-eplus.html